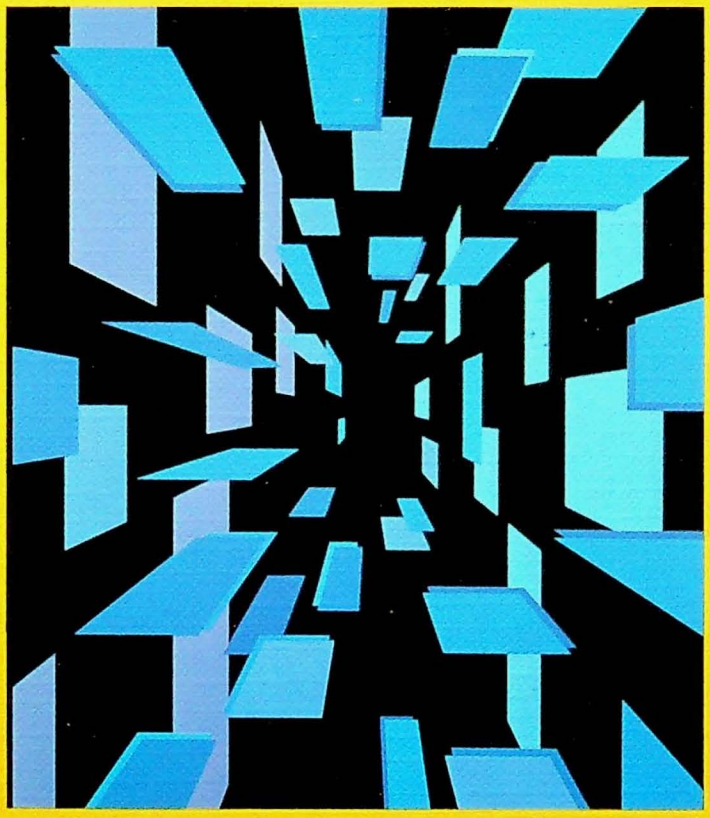
Franzis
Computer-Praxis

Zilker

Praxis des Multitasking

Parallele Programmverarbeitung durch Mikrocomputer:
Technik, Strategien und Listing



Prozeßdeskriptor
Steuerung durch
Scheduler und Dispatcher
Semaphore
Semaphoren unter BSO
Einfacher Datenpuffer
Ringpuffer
Priorisierte Puffer-
verwaltung
Prozeßkommunikation
Methoden der
Interrupt-Verwaltung
Gerätetreiber
Interrupt-Verwaltung
BSO
Fehlerbehebung
Puffern durch
Nachrichten-Queues
Speicherverwaltung
Fehlerarten

Franzis'

Zilker
Praxis des Multitasking

In der Reihe
Franzis Computer-Praxis
sind erschienen:

Benda, Mikrocomputer-Technik praxisnah
Busch, Basic für Aufsteiger
Busch, Basic für Einsteiger
Busch, Der sichere Einstieg in Pascal
Esders, Assembler-Programme zum Apple II
Esders, Das Buch zum Apple II
Feichtinger, Mit Computern steuern
Haugg, Software-Engineering
Janson, Die beiden Datenbanksysteme dBase II und III
Klein, Z-80-Applikationsbuch
Klein, Mikrocomputer selbstgebaut und programmiert
Klein, Mit HEXMON Programme entwickeln
Klein, Die Prozessoren 68000 und 68008
Klein, Was ist Pascal?
Link, Messen, Steuern und Regeln mit Basic
Merker, Hardware-Erweiterung für den Apple
Merker, Hardware-Erweiterung für den ZX 81
Miedel/Kotulla, Das große CPC-Arbeitsbuch
Piotrowski, IEC-Bus
Plate, Anwenderhandbuch CP/M-68K
Plate, Betriebssystem CP/M
Plate, Computergrafik: Einführung – Algorithmen – Programmentwicklung
Plate/Lecher, Hard- und Software für den Epson HX-20
Plate/Wittstock, Pascal: Einführung – Programmentwicklung – Strukturen
Pütz, Praxis der Datenübertragung
Pütz, Das große C-64-Handbuch
Röckrath, Mikrosoft-Basic: Konzepte, Algorithmen, Datenstrukturen
Ruhland, DOS 3.3 – das Diskettenbetriebssystem des Apple II
Schirmacher, MacIntosh programmieren
Schoffa, Die Programmiersprache LISP
Stenzel, Maschinensprache? – kein Problem!
Troitzsch, Mikrocomputer-Schaltungstechnik
Wunderlich, Erfolgreicher mit CBM arbeiten
Wunderlich, Erfolgreicher mit dem VC 64 arbeiten
Zech, Die Programmiersprache Forth

Franzis Computer-Praxis

Michael Zilker

Praxis des Multitasking

Parallele Programmverarbeitung durch Mikrocomputer:
Technik, Strategien und Listing

Mit 34 Abbildungen

Franzis'

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Zilker, Michael:
Praxis des Multitasking : parallele Programmverarbeitung durch Mikrocomputer: Technik, Strategie u. Listing / Michael Zilker. – München : Franzis, 1987.
(Franzis Computer-Praxis)
ISBN 3-7723-8561-3

© 1987 Franzis-Verlag GmbH, München

Sämtliche Rechte – insbesondere das Übersetzungsrecht – an Text und Bildern vorbehalten. Fotomechanische Vervielfältigungen nur mit Genehmigung des Verlages. Jeder Nachdruck, auch auszugsweise, und jegliche Wiedergabe der Bilder sind verboten.

Satz und Druck: Kösel, Kempten

Printed in Germany · Imprimé en Allemagne

ISB N 3-7723-8561-3

Vorwort

Betriebssysteme, die den Mehrprogrammbetrieb unterstützen, sind naturgemäß komplexe Softwareprodukte. Das folgende Buch will die Konstruktion und den Gebrauch solcher Systeme methodisch darstellen und zugleich ein kleines und kompaktes Betriebssystem für Realzeitanwendungen als exemplarisches Beispiel vorstellen.

Die dargestellten Konzepte beruhen auf allgemeingültigen Methoden, wie sie in jedem Multitasksystem zu finden sind. Dabei wurden diverse Bereiche, wie Dateiverwaltung, ausgeklammert, da die Zielanwendung, hier die kleine und mittlere Prozeßdatenverarbeitung, diese Mechanismen nicht benötigt.

Das Buch ist also für diejenigen geschrieben, die in der Praxis mit einem Mikroprozessor komplexe Echtzeitanwendungen realisieren wollen und dabei an die Grenzen der „großen Schleife" gestoßen sind. Es richtet sich an Software-Entwickler, die selber ein eigenes Betriebssystem, oder das in diesem Buch angebotene, hier für den Z80 ausgeführt, einsetzen wollen und sich damit eine Basis schaffen, die das parallele Programmieren ermöglicht und auf Dauer eine Arbeitserleichterung schafft.

Die einzelnen Kapitel teilen sich im allgemeinen in zwei Teile. Der erste Teil stellt das behandelte Gebiet theoretisch dar; im zweiten Teil wird dann das Thema konkret mit Hilfe von Z80 Mnemonics beschrieben. Aus diesem Grund ist die Kenntnis des Z80 von Vorteil, jedoch nicht zwingend zum Verständnis notwendig, da die Inhalte der ersten Kapitelabschnitte völlig unabhängig von einer Prozessorstruktur sind.

Das Buch beschreibt, beginnend mit den Grundmechanismen (z. B. Scheduling), nacheinander die benötigten Prozeduren eines Multitasksystems. Dabei bauen die Kapitel direkt aufeinander auf; es empfiehlt sich also, das Buch komplett von vorne bis hinten zu lesen.

Dieses Buch ist meiner lieben Frau Heidi gewidmet. Sie hat seine Entstehung mit angeregt und das Manuskript mit Sachverstand probegelesen. Ihr gilt besonderer Dank.

Anmerkung:

Das Betriebssystem BS0 ist auf Floppy Disk vom Autor direkt zu beziehen.

Wichtiger Hinweis

Die in diesem Buch wiedergegebenen Schaltungen und Verfahren werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden*).

Alle Schaltungen und technischen Angaben in diesem Buch wurden vom Autor mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. Der Verlag sieht sich deshalb gezwungen, darauf hinzuweisen, daß er weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernehmen kann. Für die Mitteilung eventueller Fehler sind Autor und Verlag jederzeit dankbar.

*) Bei gewerblicher Nutzung ist vorher die Genehmigung des möglichen Lizenzinhabers einzuholen.

Inhalt

1 Einleitung 9

2 Betriebssystemkern 11
2.1 Prozeßdeskriptor 16
2.2 Steuerung der Prozesse durch Scheduler und Dispatcher 19
2.3 Systemprozeduren des Kerns 21
2.4 Betriebssystemkern von BSØ 23

3 Semaphoren 28
3.1 Was ist eine Semaphore und wie setzt man sie ein? 33
3.2 Realisierung von Semaphoren 37
3.3 Semaphoren unter BSØ 42

4 Datentransport zwischen Tasks 45
4.1 Einfacher Datenpuffer 46
4.2 »No Wait« Prozeduren 48
4.3 Ringpuffer 50
4.4 Puffern durch Nachrichten-Queues 53
4.5 Priorisierte Pufferverwaltung – Pufferstrategien 55
4.6 Prozeßkommunikation mit Antwort 57
4.7 Ringpuffer unter BSØ 59

5 Interruptverwaltung 62
5.1 Interrupts schnell aber gefährlich 63
5.2 Methoden der Interrupt-Verwaltung 63
5.3 Zentrale Interrupt-Verwaltung im Systemkern 64
5.4 Gerätetreiber 66
5.5 Interruptverwaltung von BSØ 68

6 Fehlerbehandlung 71
6.1 Fehlerarten 71
6.2 Fehlerbehandlung 72
6.3 Fehlerbehandlung von BSØ 74

7 Speicherverwaltung 76

8 Programmierung unter Multitasksystemen 80
8.1 Problem der Verklemmung 80
8.2 Programmierung reentranter Prozeduren 81
8.3 Unterprogrammbibliotheken 83
8.4 Hintergrundmonitor 84

Anhang 86
BS0 Listings mit Beispieltasks 86

Sachverzeichnis 111

1 Einleitung

Mikroprozessoren begegnen dem modernen Menschen überall. Sie befinden sich in Massenartikeln und erledigen dort vielseitige Aufgaben. So steuern sie Maschinen, spielen Schach oder übernehmen Überwachungsaufgaben im medizinischen Bereich, um nur drei konkrete Anwendungen zu nennen.

Eigentlich können diese Tätigkeiten nicht allein vom Mikroprozessor ausgeführt werden, sondern sind nur möglich, wenn ein entsprechendes Programm zur Verfügung steht. Da dieses Programm das eigentliche Verhalten eines Prozessors bestimmt, ist es mitverantwortlich für das spätere korrekte Verhalten eines Mikroprozessorsystems.

Mikrorechner sind wie auch digitale Großrechner nach dem Prinzip von J. E. Neumann aufgebaut *(Abb. 1.1)*. Da die Prozessoren großer Rechner eine längere Tradition haben als Mikroprozessoren, läßt sich die Erfahrung auf dem Gebiet ihrer Programmierung auf Mikroprozessoren direkt übertragen. So hat man komplexe Softwaresysteme, sei es auch in abgemagerten Versionen, auf Mikrorechnern implementiert. Es ist dadurch möglich geworden, auch auf kleinen Systemen mit anspruchsvollen Programmiersprachen zu arbeiten. Zum Beispiel ist die Verwendung von PL/I, welche als Sprache mit sehr mächtigem Umfang bekannt ist, durchaus auf 8-Bit-Prozessoren – wenn auch vielleicht etwas unhandlich – möglich.

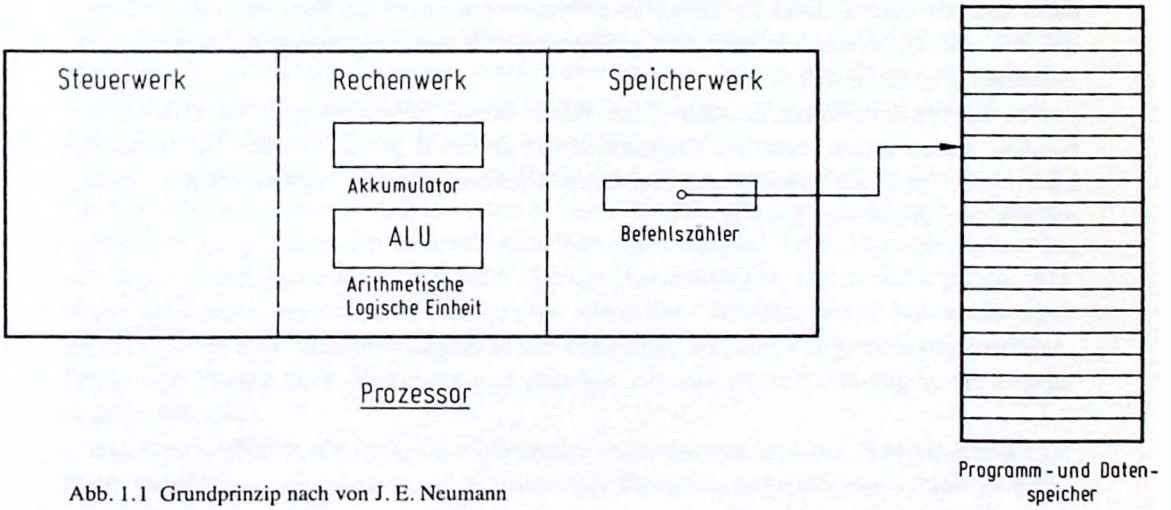


Abb. 1.1 Grundprinzip nach von J. E. Neumann

Nicht nur die Methoden von Programmiersprachen, sondern auch von anderen Software-Themen wurden auf Mikrorechner übertragen. Bekannte Betriebssysteme, wie z. B. CP/M oder MS-DOS, verhalten sich im Grunde ähnlich wie Betriebssysteme größerer Anlagen; z. B. RT-11, ein Betriebssystem der legendären PDP-11.

Eine Besonderheit stellt das System Unix dar, das direkt von größeren Anlagen auf Mikroprozessoren portiert wurde. Dies hat seinen Grund darin, daß Unix fast komplett in einer Hochsprache formuliert ist, und sich somit als portierfreudiger erweist.

Unix bietet im Gegensatz zu den obengenannten Betriebssystemen den weiteren Vorteil, daß es ein echtes Mehrprogrammbetriebssystem ist. D. h. der Rechner kann gleichzeitig mehrere Probleme bearbeiten, bzw. mehrere Benutzer bedienen. Durch parallele Programmiersysteme wird zum einen erreicht, daß der Programmierer durch grundlegende Betriebssystemfunktionen unterstützt wird (z. B. Verzögern eines Programms um eine konkrete Zeit), und zum anderen, daß eine isolierte Bearbeitung der einzelnen Prozesse erfolgt. Ferner ermöglicht ein solches System die effizientere Auswertung von einzelnen Peripheriegeräten, die im Bereich eines Multitasksystems mehreren Benutzern zur Verfügung stehen.

Wie oben erwähnt wurden auch auf diesem Gebiet die grundlegenden Ideen aus dem Bereich der Großrechner übernommen. Sie wurden dort besonders von E. W. Dijkstra und A. Hoare, die als Pioniere dieser Thematik anzusehen sind, begründet.

Bei vielen Mikroprozessoranwendungen ist ein Multitasksystem wünschenswert, jedoch ist selten ein optimales System vorhanden. Betriebssysteme wie Unix sind zwar verfügbar, aber jedoch für spezielle Zwecke (z. B. Maschinensteuerungen) aus verschiedenen Gründen ungeeignet. Diese Systeme sind in der Regel zu umfangreich, da sie Gebiete abdecken (z. B. Dateiverwaltung), die bei speziellen Anwendungen nicht gebraucht werden. Außerdem sind sie selten im ROM lauffähig und zudem mit Lizenzgebühren behaftet.

Vereinzelt werden Betriebssysteme angeboten, die diese Eigenschaften nicht haben, dafür sind sie starr an einen Prozessortyp gebunden und somit nur für einen kleinen Kreis von Anwendern interessant. Hier sei das System ZRTS von Zilog genannt, das nur für den Prozessor Z8000 erhältlich ist.

Aus dieser Lücke heraus entstand dieses Buch, dessen folgende Kapitel dem Multitask-Neuling helfen sollen, parallele Programme mit durchaus professionellen Methoden zu konzipieren und auf seinem eingefahrenen Prozessortyp zu implementieren. Dabei wünscht der Autor viel Erfolg!

2 Betriebssystemkern

Der Einsatz eines Mehrprozeßsystems ist immer dann sinnvoll, wenn mehrere Aufgaben naturgemäß parallel ablaufen können. Die Entscheidung, welche Aufgaben nebeneinander laufen dürfen, ist in der Regel durch die Betrachtung des gesamten Problems zu fällen. Dabei ist zu beachten, daß physikalisch nur ein Prozessor vorhanden ist, der gleichzeitig mehrere Aufgaben ausführen muß. Dies klingt zunächst unlogisch und ist in der Tat widersprüchlich; denn ein Prozessor arbeitet definitionsgemäß, immer sequentiell eine Liste von Befehlen (Programm), die in seinem Maschinen-Code vorliegen, ab. Er führt nie gleichzeitig mehrere Programme aus, sondern verarbeitet strikt nacheinander die vorgegebenen Programmbefehle.

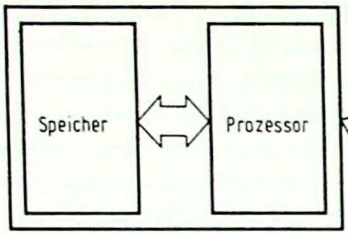
Eine korrekte Lösung des Problems wäre es, pro Aufgabe eine eigene CPU einzusetzen. Dies widerspricht jedoch unserer obengenannten Forderung nur einen Prozessor zu verwenden. Ferner ist dies mit einem Hardware-Aufwand verbunden, der in der Praxis nur in Ausnahmefällen (zeitkritische Probleme) zu vertreten und sinnvoll ist.

Ziel ist es nun, eine Methode zu finden, die es uns erlaubt, mit einem Prozessor eine gewisse Anzahl von Aufgaben gleichzeitig zu verarbeiten. Dies ist nur möglich, wenn man folgende grundlegende Kompromisse schließt: Die einzelnen Aufgaben, hier als Prozeß und verbreitet auch als Task bezeichnet, werden nicht gleichzeitig, sondern zeitlich verzahnt verarbeitet. Man spricht dann von einer quasi-simultanen oder quasi-parallelen Verarbeitung.

Durch ein geeignetes Steuerprogramm, dem sogenannten Betriebssystemkern, wird gewährleistet, daß jeder Prozeß eine bestimmte Rechenzeit zur Verfügung gestellt bekommt. Die einzelnen Aufgaben werden also reihum, jeweils für einen Zeitabschnitt, ausgeführt. Wählt man die Zeitintervalle hinreichend klein, so spiegelt man den einzelnen Benutzern eine Parallelität der Programmausführungen vor.

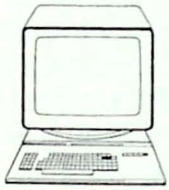
Um die Materie zu verdeutlichen, stelle ich die Lösung eines gedachten Problems vor. Ein Mikroprozessorsystem soll in einem kleinen Betrieb, für zwei Anwender gleichzeitig zur Verfügung stehen. Der eine will sein Textverarbeitungssystem, der andere hingegen ein Lagerverwaltungsprogramm fahren. Andere Anwendungen soll es nicht geben. Als Hardware existiert ein Prozessor mit Speicher sowie zwei Terminals nebst Anschlußmodulen *(Abb. 2.1)*. Aus Vereinfachungsgründen betrachten wir hier ein System ohne externe Geräte wie Floppy Disk, Festplatte und Drucker; obwohl diese Peripherie in der Praxis unentbehrlich ist.

Daneben verfügen wir über die eigentlichen Anwenderprogramme (Lagerhaltung und Textverarbeitung), die schon auf ihre jeweiligen Terminals angepaßt und fehlerfrei sind.



Lagerverwaltung

Abb. 2.1 Mikrorechner mit zwei Benutzern



Textsystem

Abb. 2.2 Speicherbelegung

Was jetzt noch fehlt ist der besagte Systemkern, der eine Parallelverarbeitung simuliert. Bevor er näher beschrieben wird, legen wir die Belegung des Arbeitsspeichers fest *(Abb. 2.2)*.

Da die meisten Mikroprozessoren nach dem Einschalten die Befehlsausführung bei Adresse 0 beginnen, plazieren wir den Systemkern dort hin, und überlassen ihm die Ausführung des Urstarts.

Die beiden Anwenderprogramme können auf beliebige Adressen des Speichers gesetzt werden. Wichtig ist dabei nur, daß beide Programme ihre eigenen exklusiven Bereiche

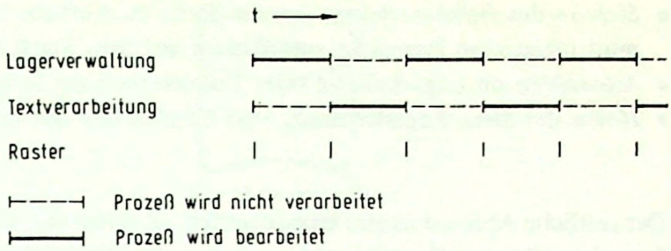


Abb. 2.3 Zeitverhalten

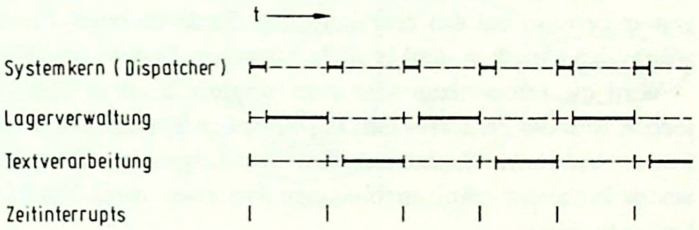


Abb. 2.4 Vollständiges Zeitverhalten

haben, und daß sich ein Programm nie auf den Speicherbereich eines anderen beziehen darf. Darüber hinaus ist eine Überlappung der einzelnen Teile vorläufig verboten.

Hierauf aufbauend kann die Leistung des Kern näher definiert werden:

Um die parallele Ausführung zu simulieren, muß er in einem gewissen Zeitraster von einer Anwendung zur anderen schalten. Das heißt Lagerverwaltung und Textverarbeitung werden wechselseitig ausgeführt *(Abb. 2.3)*. Das Zeitraster stellt sinnbildlich den Herzschlag des Systems dar. Diesen Uhrtakt erzeugt man im allgemeinen durch eine entsprechende Hardware (Timerbaustein oder Zeitglied).

Da alle gängigen Mikroprozessoren eine mehr oder minder komfortable Interrupt-Logik besitzen, liegt es hier nahe, dem Zeittakt einen solchen Interrupt auslösen zu lassen. Hierdurch wird eine korrekte Zeitsteuerung gewährleistet. Der Interrupt führt in einen Teil des Systemkerns, welcher als Dispatcher bezeichnet wird. Hier hat er im Speziellen die Aufgabe auf das andere Anwenderprogramm zu schalten.

Abbildung 2.4 zeigt nun den korrekten Zeitablauf unseres fiktiven Systems. Um zu unterscheiden, welches Anwenderprogramm als nächstes fortzuführen ist, muß der Dispatcher sich in einem eigenen Speicherbereich merken, welcher Prozeß gerade aktiv war. Ferner muß er die aktuellen Registerstände (Arbeitsregister und Befehlszähler) des soeben unterbrochenen Prozesses merken, um sie beim nächsten Interrupt wieder mit diesen Registerinhalten fortführen zu können.

Der Dispatcher hat also grob betrachtet nach einem Zeitinterrupt folgende Aufgaben:

- *Sichern* der aktuellen Registerinhalte, um den Prozeß später mit diesen Registerständen fortsetzen zu können.

- *Sichern* des Befehlszählers, um die Stelle zu merken, wo der Zeit-Interrupt zuschlug; wird meist vom Prozessor automatisch auf dem Stack ausgeführt.
- *Auswählen* ob Lagerhaltung oder Textverarbeitung fortgeführt werden soll.
- *Holen* der alten Registerinhalte und *Fortführung* des ausgewählten Prozesses.

Der zeitliche Abstand zweier Uhr-Interrupts (Zykluszeit), liegt in der Praxis im Bereich von 5 ms bis 100 ms. Es sollte abhängig von der Anwendung ein sinnvoller Wert gewählt werden. Bei unserem gedachten System wären 50 ms angemessen. Wird die Zeit zu lange genommen, so hat das eine langsame Reaktion beim Anwender zufolge; der Benutzer würde dann merken, daß er nicht allein am System beteiligt ist.

Wird die Intervallzeit sehr kurz eingestellt, so erfolgt zwar eine schnelle Reaktion, jedoch wird der Prozessor unnötig durch den Systemkern belastet, da dieser durch die hohe Interruptrate sehr oft aktiviert wird. Das aufgezeigte Beispiel läßt erkennen, wie man mit einem Prozessor zwei unabhängige Prozesse, unter Zuhilfenahme eines Uhrinterrupts, bedienen kann.

Existierende Betriebssysteme sind allgemeiner formuliert. Bei ihnen kommt es nicht darauf an, was die einzelnen Prozesse tun, sondern nur wie sie zu verwalten sind und wo sich ihr Programm befindet.

Wichtig dabei ist, daß die Anwenderprozesse auf konkreten Adressen gelinkt bzw. generiert wurden. Dem Betriebssystem teilt man dann nur noch mit, wie viele Prozesse im Gesamtsystem vorhanden sind und wo sich deren Objekt-Code befindet. Das Betriebssystem arbeitet nacheinander, jeweils in Abständen eines Taktes, die Programme ab und erzeugt so das Bild, als würden sie tatsächlich parallel ausgeführt werden. Man spricht hier von einem Time-Sharing-System, das auch oft als *Round-Robin*-Verfahren (Reihumverfahren) bezeichnet wird.

In unserem obigen Beispiel ist die gesamte Laufzeit eines Prozesses eigentlich unbegrenzt, da er ständig Prozessorzeit zugeteilt bekommen will. Ein Prozeß, der auf diese Vergabe wartet, befindet sich im Zustand „ready" (fertig zur Ausführung). Der Systemkern versetzt den Prozeß in den Zustand „running" (laufend), wenn er diesem Prozeß den Prozessor zur Verfügung stellt.

Im Beispiel pendelt der Zustand eines Prozesses ständig von „ready" nach „running" und umgekehrt. Die möglichen Zustände (bisher haben wir nur zwei kennengelernt) kann man sehr übersichtlich durch sog. Zustandsgraphen darstellen *(Abb. 2.5)*. Die Kreise symbolisieren den jeweiligen Zustand eines Prozesses; die Pfeile hingegen geben die möglichen Zustandsübergänge an. Der Übergang von einem Zustand in einen anderen wird durch die Ausführung des Dispatchers bewerkstelligt. Ein Pfeil im Zustandsgraphen stellt somit die Durchführung einer Kernprozedur dar. Bei unseren bisher eingeführten zwei Zuständen, werden die beiden Übergänge vom Dispatcher durchgeführt. Später werden wir den Graphen noch um einige zusätzliche Zustände erweitern. Durch ein weiteres Beispiel möchte ich eine andere Systemvariante aufzeigen:

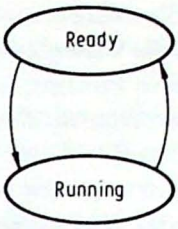


Abb. 2.5 Einfacher Zustandsgraph

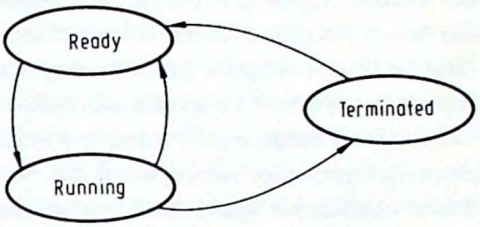


Abb. 2.6 Erweiterter Zustandsgraph

Durch das Schedule-Verfahren (schedule zu deutsch Fahrplan) können verschiedene Prozesse zu bestimmten Zeitpunkten gestartet werden. Stellen wir uns zwei Prozesse eines Überwachungsrechners mit folgenden Aufgaben vor. Der erste Prozeß soll eine Uhrzeitverwaltung durchführen und jeweils die neu entstandene Zeit, in augenfälliger Form, auf der oberen Hälfte eines Bildschirms ausgeben. Der Prozeß muß demnach, um jede Sekunde präzis registrieren zu können, im Intervall von einer Sekunde gestartet werden.

Ein zweiter Prozeß führt alle 10 s eine Temperaturmessung durch und zeigt den ermittelten Wert in der unteren Hälfte des Schirms an. Der Dispatcher aus dem vorigen Beispiel ist hierzu unverändert nicht brauchbar; er muß modifiziert werden.

Der neue Systemkern braucht die Information, in welchen Rhythmen die vorhandenen Prozesse gestartet werden sollen. Diese erhält er einmalig beim Programmstart.

Der veränderte Dispatcher erhebt einen Prozeß in den Zustand „ready", wenn eine Intervallvorgabe, durch das Auftreten der Grundtaktinterrupts, abgelaufen ist. Durch den Zustand „ready" wird der Anspruch auf Prozessorzuteilung signalisiert. Falls kein anderer Prozeß im Zustand „running" ist, so geht er in diesen über und belegt damit die CPU. Der Prozeß muß dem Kern – nach Abschluß seiner Aufgabe – mitteilen, daß er beendet ist. Der Kern seinerseits versetzt den Prozeß dadurch in den Zustand „terminated" (beendet) und gibt somit den Prozessor für andere Aufgaben frei. Der Zustand „terminated" unterscheidet sich gegenüber „ready" dadurch, daß ein Prozeß der „terminated" ist keine CPU-Zeit beansprucht; man könnte sagen, der Prozeß schläft. *Abbildung 2.6* zeigt den neuen Graphen. Der Zustand „terminated" ist der Ruhezustand eines Prozesses. Dort befindet er sich zu Beginn, bevor er durch irgendein Ereignis „ready" wurde, und nach seiner Ausführung, wenn er einen Durchlauf hinter sich hat.

Ein Prozeß gelangt von „terminated" über den Scheduler, der die Zeitverwaltung erledigt, nach „ready". Auch bei dieser Methode ist es möglich, daß mehrere Prozesse zur

gleichen Zeit „ready" werden. Eine sinnvolle Abwicklung wäre hier das erwähnte Round Robin Verfahren oder auch die Strategie: wer zuerst „ready" wird, wird als erster komplett bedient (first come first serve Prinzip), ist gebräuchlich.

Ist niemand „ready", so führt der Dispatcher auf einen systeminternen Prozeß, der definiert nichts tut (Idle Prozeß).

Bei gängigen Betriebssystemen wird des weiteren die Möglichkeit der Prioritätenvergabe geboten. Eine Steuerung des Prozeßganges nach Prioritäten ist dann sinnvoll, wenn die Ausführung eines Prozesses wichtiger ist, als die Ausführung eines anderen. Prozesse mit hoher Priorität werden schneller bearbeitet als die mit niederer Priorität. Bei ihrer zahlenmäßigen Angabe hat sich eingebürgert, daß sie mit fallendem Wert steigt. Ein Prozeß mit Priorität 10 kommt schneller an den Prozessor als ein Prozeß mit Priorität 30.

Die bisher aufgezeigte Prozeßverwaltung, stellt die Wirkungsweise des innersten Kerns eines Mehrprozeßbetriebssystems dar. Sie ist in dieser Weise allgemeingültig. Ausgehend davon werden wir später noch zusätzliche Elemente um den Kern gruppieren, die einen erweiterten Funktionumfang darstellen. Dabei handelt es sich um Interrupt-Verwaltung, Ein-, Ausgabesteuerung und dynamische Speicherverwaltung.

2.1 Der Prozeßdeskriptor

Wie jedes Programm besteht auch der Systemkern aus einem Code-Teil und einem Datenbereich. Der Code-Teil beinhaltet die auszuführenden Befehle, wie zum Beispiel die des Dispatchers, und bezieht sich dabei auf Datensegmente, die für die Prozeßverwaltung nötig sind.

Ungeachtet einer Implementierungssprache folgt eine detaillierte Beschreibung des Datenbereiches von Dispatcher und Scheduler.

Wie bereits erwähnt wurde, muß der Systemkern über jeden einzelnen Prozeß buchführen, d. h. er speichert Daten über ihn; z. B. den aktuellen Zustand.

Diese Angaben werden in einem Datenblock – von dem pro Prozeß einer vorhanden ist – gespeichert. Die unerläßlichen Elemente dieses *Prozeßdeskriptors* sind:

- Stackpointer des prozeßeigenen Stacks
- aktueller Prozeßzustand („ready", „running", usw.)
- Startadresse des Prozesses.

Es ist sinnvoll, jedem Prozeß einen eigenen Stackbereich zuzuteilen, da die Prozesse dadurch datenmäßig entkoppelt werden. Dies ist wesentlich, weil zum einen die Prozesse dadurch eigene Datenwerte dynamisch ablegen können und zum anderen dort der Dispatcher, der nach jedem Zeitinterrupt aktiviert wird, die aktuellen Registerstände des unterbrochenen Prozesses sichert.

Das zweite Element des Prozeßdeskriptors beherbergt den aktuellen Zustand eines Prozesses. Hier wird einer der Zustände (bisher bekannt: „ready", „running" und „terminated") vom Kern verzeichnet.

Im dritten Element findet der Systemkern die Anfangsadresse des betreffenden Prozesses. Dorthin verzweigt der Dispatcher, wenn er den Prozeß erstmalig bzw. erneut anlaufen lassen will.

Mit diesen drei Elementen läßt sich durchaus ein für einfache Anwendungen brauchbares Multitasksystem aufbauen. Im Fall einer priorisierten Verwaltung der Prozesse, was allgemein üblich ist, baut man aus einzelnen Deskriptorblöcken eine verkettete Liste auf (*Abb. 2.7*a). Dabei hat der Prozeß, dessen Deskriptor am Anfang der Kette ist, die höchste, und der zuletzt eingekettete Prozeß die niederste Priorität. Auf diese Weise ist es weiterhin möglich, die Priorität eines Prozesses dynamisch zu verändern. Dazu ist nur die Modifikation dreier Zeiger nötig. Abb. 2.7b zeigt dies für die Kette 1-3-2.

Die Bezeichnung NIL bedeutet, daß hier das Ende der Kette ist; es folgt kein weiterer Deskriptorblock. Der Anker hingegen ist ein Zeiger, der auf den ersten und somit auf den höchstpriorisierten Prozeß zeigt. Der Ankerzeiger selbst gehört nicht zu irgendeinem Deskriptorblock, sondern zum Dispatcher.

Um auch das vorgestellte Schedule-Verfahren mit einzubringen, ist die Erweiterung des Deskriptorblocks nötig. Dafür werden zwei zusätzliche Elemente gebraucht.

- Aktueller Zähler
- Intervallzeitvorgabe

Nach jedem Zeit-Interrupt wird erst der Scheduler und danach der Dispatcher durchlaufen. Der Scheduler verwaltet pro Aufruf nacheinander die jeweiligen aktuellen Zähler der Tasks. Falls er größer als Null ist, wird er vom Scheduler dekrementiert.

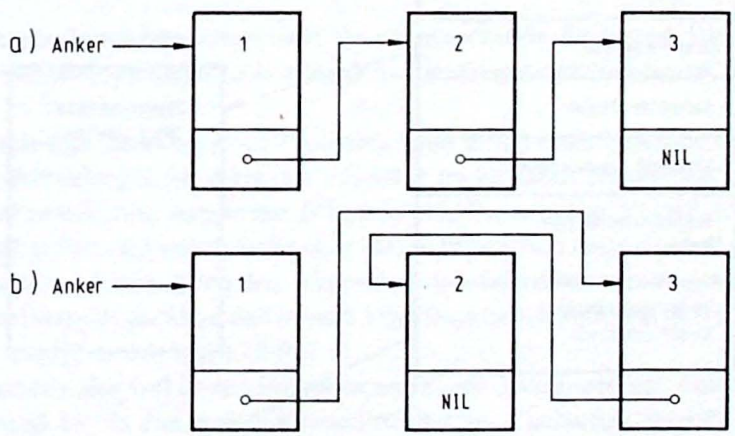


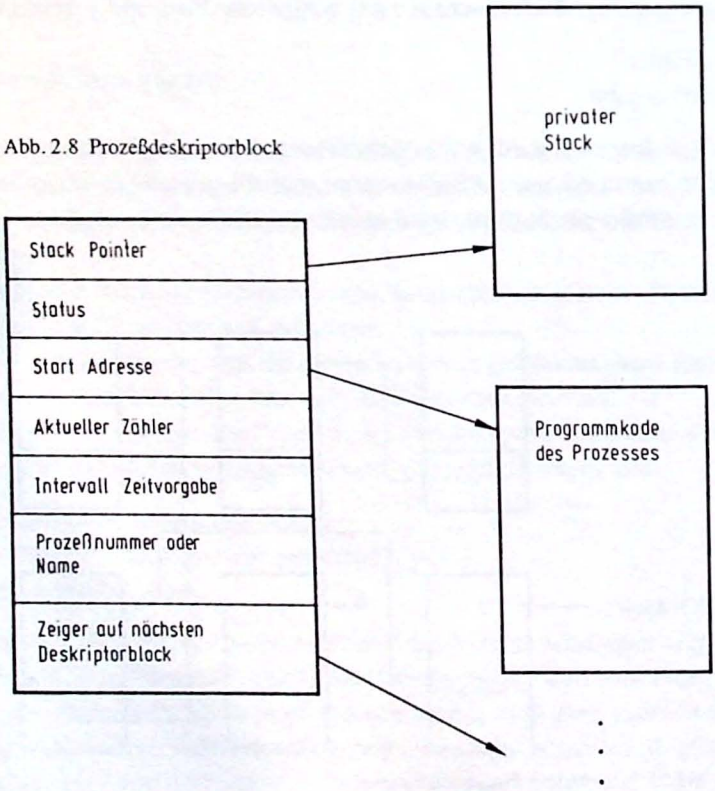
Abb. 2.7 Verkettete Prozeßdeskriptoren

Wir arbeiten hier mit einem Abwärtszähler. Springt der Zähler von 1 auf 0 über, bringt der Scheduler den Prozeß in den Zustand „ready". Beim nächsten Interrupt wird der aktuelle Zähler dann mit der Intervallzeit neu vorgeladen. Jeder weitere Zeitinterrupt dekrementiert den aktuellen Zähler und setzt den Prozeß, der inzwischen abgearbeitet wurde, beim Übergang auf Null wieder „ready". So ist der zyklische Start von Prozessoren sehr präzise und einfach zu realisieren.

Erinnern wir uns an das Beispiel des Überwachungsrechners, auf dem zwei Prozesse zyklisch laufen sollen. Bei der Annahme legen wir einen Grundtakt von 10 ms zugrunde. Somit muß auch dem Prozeß, der alle Sekunde laufen soll, der aktuelle Zähler und die Intervallzeitvorgabe auf den Wert 100 gesetzt werden. Durch den Scheduler wird der Prozeß erstmalig nach einer Sekunde auf „ready" gesetzt und im späteren Verlauf ausgeführt. Durch laden des aktuellen Zählers mit der Intervallzeitvorgabe wird der nächste Start (1 Sekunde später) vorbereitet.

Eine wichtige Ergänzung des Deskriptorblocks ist die Einführung einer Prozeßidentifikation. Sie besteht entweder aus einer Nummer oder aus einem Namen. Auf sie kann nicht verzichtet werden, wenn man das System über spezielle Kernprozeduren beauftragen will,

Abb. 2.8 Prozeßdeskriptorblock



einen Prozeßlauf in irgendeiner Weise zu beeinflussen. Als Beispiel sei die Änderung der Priorität oder die zwangsweise Suspendierung eines Prozesses genannt. Diese Prozeduren werden im Teilkapitel 2.3 behandelt. Man gibt bei ihrem Aufruf in einer bestimmten Weise die Nummer oder den Namen an, um ihr mitzuteilen, auf welchen Prozeß sie sich beziehen soll. Die Kernroutine sucht sich dann selbständig den entsprechenden Deskriptorblock aus und modifiziert ihn entsprechend ihrer Aufgabe. *Abbildung 2.8* zeigt den Aufbau eines vollständigen Prozeßdeskriptors.

Allgemein beinhaltet er alle Informationen, die sich auf einen konkreten Prozeß beziehen. Hier müssen bei Bedarf eventuelle Zugriffsrechte auf Geräte verzeichnet oder weitere Prozeßkennzeichen angeführt werden.

2.2 Steuerung der Prozesse durch Scheduler und Dispatcher

Die wichtigste Prozedur des Systemkerns ist zweifellos der Dispatcher. Er allein entscheidet, ob ein Prozeß gestartet bzw. fortgeführt wird, oder ob er weiterhin „schlafen" soll.

Seine Aktionen sollen nun, in einem folgenden Beispiel, näher präzisiert werden. Durch eine verkettete Liste, so wie in Abb. 2.7, soll eine Prioritätensteuerung vorliegen. Nach dem Zeit-Interrupt wird zunächst ein Abschnitt zur Registersicherung sowie der Scheduler durchlaufen, danach hat der Dispatcher die Aufgabe, den Prozeß mit höchster Priorität fortzuführen.

Es handelt sich dabei um den Prozeß, der als erster, in der Kette der Deskriptoren, den Zustand „ready" trägt. Verhalten sich die Stati wie in *Abb. 2.9*a, so würde der Dispatcher den Prozeß 2 fortführen. Für ihn sind also nur Prozesse interessant, die im Zustand „ready" sind. Bei weiteren Zeit-Interrupts würde der Dispatcher nun immer den Prozeß 2 fortführen. Deshalb ist bei diesem Steueralgorithmus dafür Sorge zu tragen, daß sich ein Prozeß, der seine Aufgabe erledigt hat, selber beendet; d. h. er begibt sich in den Zustand „terminated".

Diesen Algorithmus bezeichnet man als unterbrechende Steuerung (ein Prozeß kann zugunsten eines anderen unterbrochen werden). Der Gegensatz dazu ist das Round-Robin-Verfahren.

Das selbständige Beenden eines Prozesses kann durch eine Systemprozedur erfolgen (Kap. 2.3). Abbildung 2.9b zeigt den Prozeß 2 im Zustand „terminated" (beendet). In dieser Statuskonstellation vergibt der Scheduler den Prozessor an Prozeß 3. Dies führt er solange fort, bis Prozeß 3 sich beendet oder bis ein höherpriorisierter Prozeß „ready" wird. Dies kann unter anderem durch den vorgeschalteten Scheduler verursacht werden.

Im Beispiel wäre es denkbar, daß Prozeß 1 zyklisch laufen soll, und durch das Ablaufen seiner Zeit „ready" würde (Abb. 2.9c).

Der Dispatcher, der vorher den Prozessor an Prozeß 3 vergeben hat, unterbricht diesen und läßt Prozeß 1 – da dieser eine höhere Priorität hat – anlaufen. Prozeß 3 kommt erst wieder in den Genuß des Prozessors, wenn Prozeß 1 „terminated" ist.

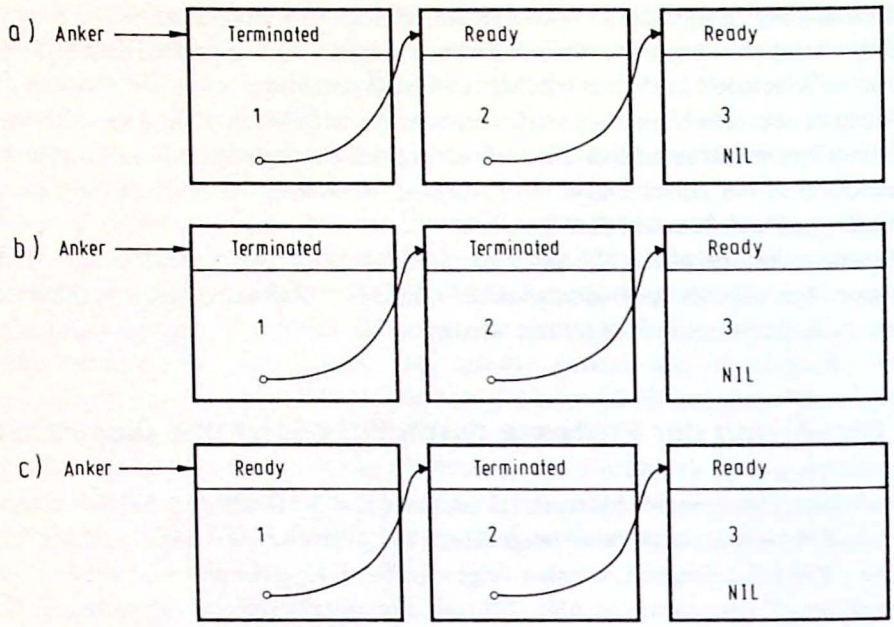


Abb. 2.9 Zeitliche Änderung der Zustände

Die unterbrechende Steuerung wird oft in der Echzeitverarbeitung angewandt. Ungeeignet dagegen wäre sie bei einem Mehrbenutzersystem. Dort nämlich laufen die einzelnen Prozesse sehr lange, und zudem können hier Arbeiten, die das System blockieren würden (eine Endlosschleife bei der Programmentwicklung), ausgeführt werden. Ein hochpriorisierter Benutzer könnte durch einen Programmierfehler alle Prozesse unter ihm lahmlegen. Hier ist Round Robin sinnvoll, denn es gibt bei diesem Verfahren keine Prioritäten, sondern die Prozesse werden nacheinander – ungeachtet ihrer Stellung in der Kette – bedient. Es sind jedoch dabei auch nur die Prozesse mit dem Zustand „ready" interessant. Eine weiterhin wesentliche Prozedur des Systemkerns ist der Scheduler. Auch er wird nach jedem Zeit-Interrupt aktiv und verwaltet die Zeitsteuerung. Er durchläuft die Prozeßdeskriptorliste von Anfang bis Ende, dabei spielt für ihn die Priorität keine Rolle. Er hat die Aufgabe – für jeden Prozeß getrennt – eine Uhr (Zähler) zu führen, und nach Ablauf dieses Zählers den entsprechenden Prozeß „ready" zu setzen. Alle Zeitangaben werden hier in Grundtakten gemacht. Bei Bedarf kann es auch sinnvoll sein, eine Verwaltung nach Uhrzeit evtl. auch Datum einzuführen. Beim Scheduler wie auch beim Dispatcher ist es wichtig, daß diese Routinen sehr kleine Durchlaufzeiten besitzen. Das spielt besonders dann eine Rolle, wenn der Grundtakt hoch ist.

Vor dem Scheduler existiert noch eine kleine Prozedur, die den Zweck hat, alle Registerinhalte eines unterbrochenen Prozesses zu sichern. Wichtig ist, daß die Inhalte der Arbeitsregister (Akku etc.) auf dem prozeßeigenen Stack abgelegt werden. Hier wird auch der Befehlszähler, der die Stelle angibt, wo der aktuelle Prozeß unterbrochen wurde, abgelegt. Der Wert des so positionierten Stack Pointers wird im Prozeßdeskriptor festgehalten. Der Systemkern schaltet, bevor er zum Scheduler und Dispatcher verzweigt, den Stack Pointer auf einen eigenen Stackbereich, über den er exklusiv verfügt, um.

Soll am Ende des Dispatchers ein Prozeß fortgeführt werden, so ist der gesicherte Stackpointer, der im ausgewählten Prozeßdeskriptor steht, Aufhängepunkt für diese Aktion. Mit ihm nämlich, können die restlichen Register auf ihren ursprünglichen Stand gebracht werden.

Bei der Ausarbeitung eines Mehrprozeßbetriebssystemes, stellt sich früher oder später die Frage nach der Implementierungssprache. Dabei sollte man sich, gerade im Umfeld von Mikroprozessoren, genau überlegen, welche Sprache geeignet erscheint. Die Kernpartien Dispatcher und Scheduler werden oft in Assembler formuliert, da an diese Teile des Systems hohe Anforderungen bezüglich der Ausführungszeit gestellt werden. Diese Prozeduren, besonders der Dispatcher, müssen Maschinenbefehle ausführen, die in einer Hochsprache in der Regel nicht verfügbar sind. Zudem muß beachtet werden, daß der Objektkode, der durch einen Compiler generiert wurde, oft sehr groß ausfällt. Gerade dies kann bei einem beschränkten Adreßraum von 64 KByte zu Speicherengpässen führen. Trotzdem sollte man auf jeden Fall den Einsatz einer Hochsprache ins Kalkül ziehen und in diesem Zusammenhang seine verfügbaren Compiler auf die genannten Merkmale prüfen.

2.3 Systemprozeduren des Kerns

Es handelt sich hierbei um Routinen, die von einzelnen Prozessen her aufgerufen werden können. Mit ihnen ist es möglich, auf den Prozeßablauf Einfluß zu nehmen.

Beenden eines Prozesses

Im Kern muß sich eine Stelle befinden, wo sich ein laufender Prozeß selber beenden kann. Man kann dies erreichen, indem man diese Stelle direkt anspringt, oder sich durch einen Return-Mechanismus dorthin begibt. Die augenfälligste Aktion wird dort sein, daß im entsprechenden Prozeßdeskriptor der Zustand „terminated“ eingetragen wird. Zusätzlich muß der Stack Pointer sowie im prozeßeigenen Stack die Startadresse restauriert werden. Zusammenfassend kann man sagen, daß hier eine Initialisierung des Prozeßdeskriptors, für den nächsten Neustart, erfolgt.

Verzögerung eines Prozesses

Mit einer DELAY-Prozedur kann sich ein Prozeß für eine gewisse Zeit verzögern. Sie wird dann gebraucht, wenn ein Prozeß für eine konkrete Zeit warten muß. Als Beispiel wäre das Abwarten einer Tastenprellzeit zu nennen.

Forderung ist, daß während des Wartens der Prozessor anderen Prozessen zur Verfügung steht. Demnach darf nicht verzögert werden, indem man Prozessorzeit durch Zählschleifen vergeudet. Man hüte sich vor solchen Methoden im Zusammenhang mit Multitasksystemen. Das Warten eines Prozesses kann ähnlich realisiert werden wie das bereits besprochene Scheduling. Es wird ein neuer Zustand, mit der Bezeichnung „delayed", eingeführt. Der entsprechende Graph ist in *Abbildung 2.10* aufgezeigt. Der Übergang von „running" nach „delayed" erfolgt durch die Prozedur DELAY. Die anderen Übergänge, bis auf die Beendigung eines Prozesses, werden durch den Dispatcher ausgeführt. Die DELAY-Routine benötigt im Prozeßdeskriptor einen zusätzlichen Zeitzähler. Er wird mit einem vom Prozeß übergebenen Wert belegt. Die Verwaltung dieses Zählers fällt in das Aufgabengebiet des Schedulers, der die Zeitvorgabe dekrementiert und den Prozeß, der sich im Zustand „delayed" befindet, nach Ablauf seiner Zeit wieder „ready" setzt. Hierdurch wird er fortgesetzt.

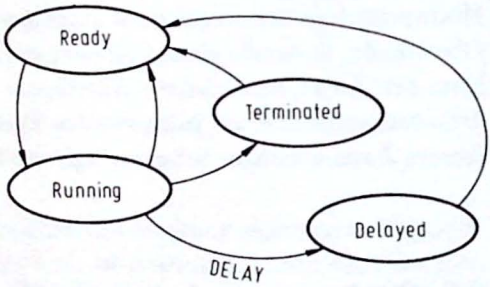


Abb. 2.10 Zustandsgraphen für System mit DELAY Funktion

Aktivieren eines Prozesses

Oft ist es nötig von einem Prozeß aus einen anderen zu aktivieren; besonders dann, wenn die Prozesse voneinander zeitlich abhängen. Die Prozedur ACTIVE ermöglicht erstmalig eine gewisse primitive Verständigung unter einzelnen Tasks. Verfeinerte Kommunikationstechniken werden später ausführlich behandelt. ACTIVE sucht zunächst den Deskriptor aus, auf den sich der Aktivierungsbefehl bezieht. Dort wird der Zustand „ready" eingetragen und so der Prozeß aufgeweckt. Die Übergabe des Namens bzw. der Nummer des betreffenden Prozesses erfolgt durch einen Parameter beim Aufruf.

Prozeßgenerierung

Beim Hochfahren des gesamten Systems muß dem Kern einmal mitgeteilt werden, welche Prozesse zur Verarbeitung anstehen. Es ist möglich, daß man abgetrennt vom Kern, die

Liste der Prozeßdeskriptoren, einschließlich ihrer Verkettung, aufbaut, und dem Kern nur eine Ankeradresse, die auf den ersten Deskriptor weist, übergibt. Dies führt zu einem besonders kleinen und einfachen Kern. Viele Betriebssysteme verwenden hingegen eine CREATE-Prozedur, die es ermöglicht von außen her Prozesse dem Kern bekannt zu machen. Sie produziert Deskriptorblöcke und hängt diese an die entsprechende Stelle der Liste ein. In einem Parameterblock werden die Prozeßkenndaten (Startadresse, Prozeßnamen, Zykluszeiten etc.) übergeben. Der große Vorteil dabei ist, daß man Prozesse bei Bedarf dynamisch generieren, und durch eine entsprechende Prozedur wieder löschen kann. Ferner ist die Schnittstelle zum Einketten von Prozessen transparenter, was bei der Systempflege eine große Rolle spielt.

Die erste Vorgehensweise wird durch die Verwendung von Daten mit strukturiertem Typ (z. B. records) und deren Verbindung durch Pointer vereinfacht; auf Assemblerebene können solche Ketten leicht mit Macros produziert werden. Andererseits ist es gefährlich, solche bedeutungsvolle Datenbereiche, wie die Deskriptoren, in die Verantwortung von außenstehenden Modulen zu geben.

Bei allen Kernprozeduren, die von Prozessen aus aufgerufen werden, ist es wichtig, den Interrupt während ihrer Ausführung zu sperren. Dadurch wird vermieden, daß der Zeit-Interrupt in einer Kernprozedur zuschlägt, was fatale Folgen haben könnte.

Die obigen Prozeduren gehören zum Standardrepertoire eines Multitasksystems und sollten generell implementiert werden. Darüber hinaus können Routinen, wie z. B. zur Verzögerung eines Prozesses bis zum Eintreffen eines Interrupts oder zur Suspendierung einzelner Prozesse usw., hier in ähnlicher Weise eingegliedert werden.

2.4 Betriebssystemkern von BSØ

BSØ ist ein Multitasksystem, welches für die Echtzeitverarbeitung von technischen Steuer- und Regelaufgaben (Prozeßdatenverarbeitung) geschaffen wurde. Diese Anwendungen sind in aller Regel zeitkritisch und erfordern eine schnelle Antwortzeit des Gesamtsystems. Aus diesem Grunde wurde BSØ komplett in ZØ Assembler kodiert. Als Nebeneffekt ist die Kompaktheit (im Vollausbau ca. 3 KByte Maschinenkode) bemerkenswert.

Im Zusammenhang mit diesem Buch lassen sich anhand des Quellkodes sämtliche beschriebenen Betriebssystemfunktionen bis auf den letzten Befehl herunter nachvollziehen. Dabei soll es hier nur aufzeigen, wie ein spezielles Betriebssystem arbeitet; es ist lediglich als Vorschlag gedacht. Jedem Leser steht es offen, Teile zu übernehmen, neu zu schreiben oder zu ergänzen. Unter BSØ wird ein Prozeß als Task bezeichnet. Die grundlegenden Verarbeitungsmerkmale gehen mit den in Kapitel 2 beschriebenen konform. Es wurde die reine Form der unterbrechenden Steuerung implementiert; ein Round-Robin-Verfahren ist nicht vorgesehen. Der Grundtakt wird mit 1Ø ms gefahren, kann aber bei Bedarf verändert werden.

Der Anwender (hier im Sinne des Programmierens der Tasks) muß eine Ankeradresse bereitstellen, die das System mit den einzelnen Taskdeskriptoren versorgt. Eine CREATE Funktion existiert nicht. Diese Ankeradresse heißt BASIS und zeigt auf eine zentrale Tabelle, die in einer Baumstruktur auf die einzelnen Deskriptoren der Prozesse verweist *(Abb. 2.11)*. Das erste Byte, auf das BASIS zeigt, gibt an, wie viele aktuelle Prozesse eingekettet sind. Danach folgen Pointer auf die Prozeßdeskriptoren der einzelnen Tasks. Ihre Reihenfolge gibt die Priorität an! Dabei ist der erste höher priorisiert als die folgenden. Die Priorität wird also nicht numerisch angegeben, sondern ergibt sich aus der Reihenfolge der Adressen auf die Taskdeskriptoren. Jeder Deskriptor enthält folgende Elemente.

- Privater Stack Pointer
  Er zeigt auf einen Speicherbetrieb, der als exklusiver Stack für einen Prozeß dient. Auffallend ist hier, daß er mitten in den Stack zeigt. Dahinter steht folgende Logik: Der Dispatcher macht keinen Unterschied zwischen erstmaligem Start und Fortsetzung eines Prozesses. Das bedeutet, daß beim Neustart einer Task, nach dem Umschalten des CPU Stack Pointers auf den Prozeßstack, zunächst sämtliche Arbeitsregister vom Stack geholt

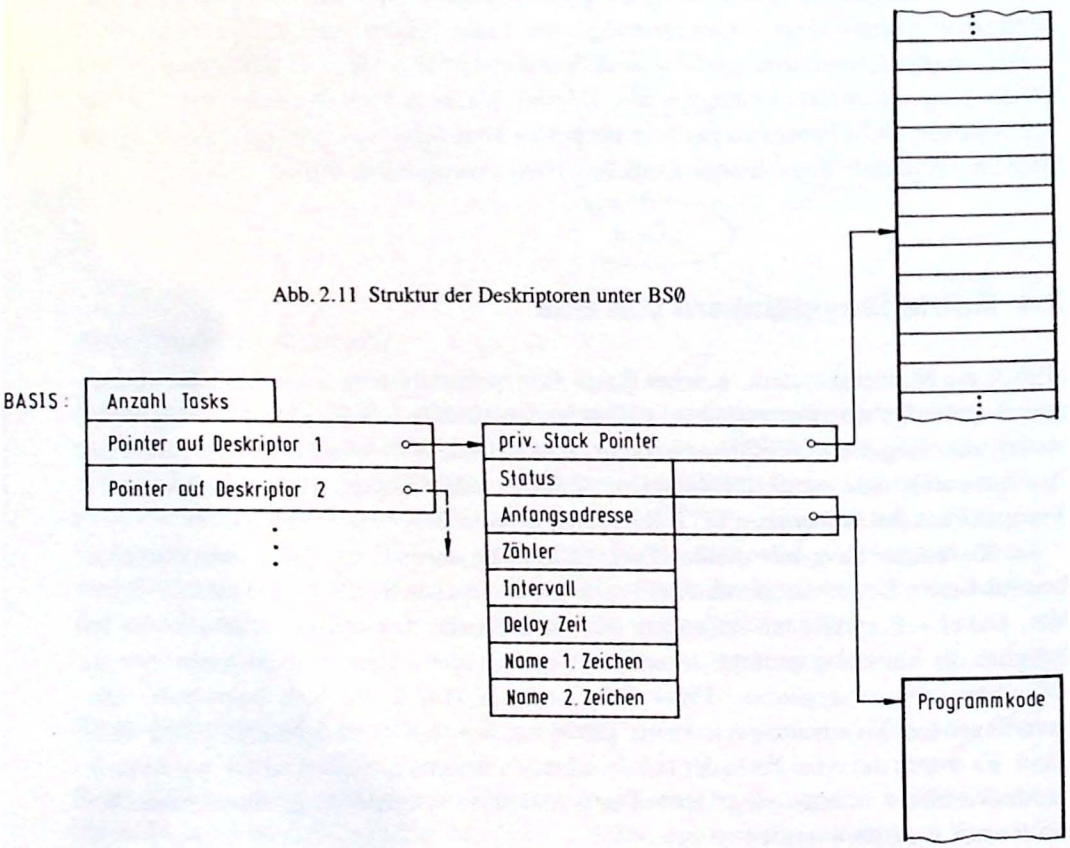


Abb. 2.11 Struktur der Deskriptoren unter BSØ

werden. Durch einen Return Befehl (RET), der die „Rücksprungadresse" vom Stack holt, wird nun die Task direkt gestartet. Der nächste Zeit-Interrupt legt dort wieder die Adresse ab, bei der der Prozeß unterbrochen wurde.

- Status
  Dieses Byte enthält den aktuellen Task-Zustand, wie bereits ausführlich diskutiert, und wird auf „terminated" initialisiert.
- Anfangsadresse der Task
  Beim zyklischen Betrieb einer Task muß ihre Anfangsadresse – nach jedem Durchlauf durch die Prozedur ENDE im privaten Stack wieder initialisiert werden.
- Zeitzähler
  Er wird für die Verwaltung von zyklischen Starts gebraucht.
- Intervall Wert
  Durch eine gezielte Initialisierung von Zeitzähler und Intervall lassen sich verschiedene Startmöglichkeiten einer Task erzielen.

Zeitzähler	Intervall	
Ø	n	Start sofort nach Urstart; danach alle n Grundtakte.
n	n	Erstmaliger Lauf nach n Grundtakten; dann zyklisch.
1	Ø	Aktivierung einmalig nach erstem Takt; dann nie mehr durch den Scheduler.
n	Ø	Start nach n Takten; danach nie mehr.
Ø	Ø	Task wird vom Scheduler nie aktiviert.

- Delay Zeit
  Dieses Byte wird von der DELAY-Funktion verwendet, um eine Task n Takte zu verzögern. Sollte mit Null initialisiert werden.
- Name
  Hier fügt sich ein Taskname, bestehend aus genau zwei Zeichen, an. Er kann beliebig vergeben werden und dient zur Identifikation einer Task.

Die gesamte Baumstruktur muß dem Systemkern, komplett initialisiert, übergeben werden. Dies legt große Verantwortung auf den Anwender, spart jedoch eine CREATE-Funktion ein.

Der Urstart des Systems erfolgt bei der Marke POWUP (Power Up). Dort wird auch die Initialisierung der Interrupt-Logik und der Bausteine, die für den Zeitinterrupt verantwortlich sind, durchgeführt. An dieser Stelle müßten gegebenenfalls noch weitere Bausteine (Interruptcontroler ect.) initialisiert werden.

Nach diesen Aktionen läuft der Prozessor auf ein HALT auf; er wartet hier auf den ersten Zeit-Interrupt. An dieser Stelle befindet sich der Prozessor, wenn keine Task „ready" oder „running" ist!

Der Zeit-Interrupt setzt bei der Marke TICK ein, und verzweigt hier in den Scheduler und später in den Dispatcher (Marke SELTSK).

Mit Hilfe des Systemeingangs ENDE muß sich eine Task abmelden. Dies erfolgt durch einen Sprung zu diesem Label.

```
...
JP ENDE                ; Beenden eines Prozesses
```

Durch die Prozedur DELAY kann sich eine Task, wie in Kapitel 2.3 beschrieben wurde, für eine bestimmte Zeit verzögern.

Mit ACTIVE kann ein Prozeß einen anderen „ready" setzen und damit starten. Der Name wird im Registerpaar BC übergeben.

Dem Z80-Kenner wird aufgefallen sein, daß in den Macros zur Sicherung der Register nie die Hintergrundregister mit verarbeitet werden. Dies wurde aus Vereinfachungsgründen weggelassen und ist bei Bedarf nachzurüsten.

Zusammenfassung

Der zentrale Dreh- und Angelpunkt eines Mehrprozeßsystems ist der sogenannte Betriebssystemkern. Bildlich gesehen stellt er den Mittelpunkt des gesamten Systems dar und besitzt demnach die höchste Verwaltungsmacht. Nur er allein kann Tasks starten und gestartete Tasks unterbrechen bzw. beenden oder die Tasks in sonstige, im Detail hier beschriebene Zustände bringen. Von außen her wird er über ganz klar definierte Schnittstellen, die z. B. angeben können, daß sich ein Prozeß um eine angegebene Zeit verzögern will, aufgerufen. Zentrale Aufgabe des Kerns ist die Vergabe der CPU an eine bestimmte Task. Welcher Prozeß behandelt wird, entscheidet der Kern anhand von Vorgaben des Programmierers. Grundsätzlich gehen wir hier davon aus, daß alle Tasks gleichzeitig im Arbeitsspeicher des Prozessors existieren bzw. geladen sind.

Der Systemkern vergibt nun die CPU an eine Task, indem er sich selber ausblendet und mit der CPU in den Programmcode einer Task geht. Dieses spezielle Task wird nun ausgeführt, und man sagt, sie ist im Zustand „running". Die Teile, die diese Aufgaben übernehmen, werden als Scheduler und als Dispatcher bezeichnet. Dabei führt der Scheduler die Zeitsteuerung und der Dispatcher die Prozessorvergabe durch. Die hierzu nötige Information über den Prozeß, wird im Prozeßdeskriptor verwaltet.

Sehr anschaulich werden die einzelnen Zustände, wenn man den Zustandsgraphen betrachtet. Er zeigt an, durch welche Ereignisse ein Prozeß von einem Zustand in den anderen gelangt (Abb. 2.6).

Durch die Vergabe der CPU an die Task hat sie eigentlich die volle Gewalt über den Rechner bekommen. Es muß nun gewährleistet werden, daß die so befähigte Task gelegentlich wieder unter die Kontrolle des Kerns gerät. Dies ist nur durch einen Interrupt sinnvoll zu realisieren. Ihn läßt man zyklisch auftreten und erhält somit einen Grundtakt. Die Auslösung des Interrupts erfolgt hardware-mäßig mit geeigneten Mitteln. Durch diese Methode wird sichergestellt, daß der Systemkern zyklisch entscheiden kann, welche Task als nächste die CPU erhält. Die Zykluszeit des Taktes sollte den Gegebenheiten der Gesamtanwendung angepaßt sein. Ein typischer Wert ist 20 ms.

3 Semaphoren

Bisher wurde aufgezeigt, wie es möglich ist, mit nur einem Prozessor mehrere Aufgaben quasi zur gleichen Zeit abzuwickeln.

Dabei wurde vorausgesetzt, daß unter den einzelnen Prozessen keine Verbindung besteht, und daß jeder Prozeß grundsätzlich verschiedene Themen, die untereinander keinen Bezug haben, bearbeitet. Das kommt der Situation gleich, als würde jeder Prozeß auf einem eigenen Rechner ausgeführt werden.

Die Anforderungen an ein Multitasksystem sind aber weitreichender, denn es wird ein Konzept benötigt, welches die Beeinflussung eines Prozesses durch einen anderen ermöglicht.

Dadurch wird erreicht, daß ein Prozeß für den Ablauf eines anderen mit verantwortlich ist. Diese Prozesse sind somit gekoppelt und werden als kooperierende Prozesse bezeichnet.

Eine klassische Problematik stellt hier die Konkurrenz um Betriebsmittel dar. Unter Betriebsmittel sind periphere Geräte, wie Drucker, Plotter oder Magnetplatte, und interne spezielle Einheiten eines Rechners wie batteriegepufferte Speicherbereiche und Spezialprozessoren zu verstehen. Diese Betriebsmittel können immer nur von einem Prozeß exklusiv benutzt werden – gemischte Verwendung ergäbe keinen Sinn. Um sie aber nun auch mehreren unterschiedlichen Prozessen zugänglich zu machen, muß ein Weg gefunden werden, der dies erlaubt.

Eine gängige Lösung ist die, daß ein Prozeß, der z. B. den Drucker verwenden will, sich zunächst beim Betriebssystem darum bewirbt. Der Systemkern entscheidet dann, ob der Prozeß das Betriebsmittel benutzen darf oder nicht. Wird ihm dies gestattet, so kann der so autorisierte Prozeß den Drucker uneingeschränkt nutzen. Ist er mit seinen Ausgaben fertig, muß er dies seinerseits dem System mitteilen, damit es das Betriebsmittel an andere bedürftige Prozesse vergeben kann.

Benutzt indessen ein anderer Prozeß den Drucker, so muß der zweite Prozeß so lange warten, bis der erste das Betriebsmittel wieder freigegeben hat. Durch eine solche Verwaltung ist garantiert, daß einerseits teuere Geräte formal allen Prozessen zur Verfügung stehen, und andererseits ein gleichzeitiger Mehrfachzugriff, welcher zu unsinnigen Ergebnissen führen würde, ausgeschlossen ist.

Synchronisation von Prozessen durch Flags

Konkretisieren wir an einem bereits aufgezeigten Fall das Problem und dessen Lösung, indem wir das Beispiel des Überwachungsrechners aufgreifen und seine Funktion dahinge-

hend ändern, daß die Ausgaben nicht auf dem Bildschirm, sondern auf einem angeschlossenen Drucker erfolgen sollen.

Benutzt ein Prozeß den angeschlossenen Drucker, so muß dies dem anderen Prozeß signalisiert werden. In nicht parallelen Systemen verwendet man zu ähnlichen Zwecken (um z. B. anzuzeigen, daß etwas frei oder belegt ist) sogenannte Flags. Diese Flags können gesetzt sein oder nicht und zeigen dadurch an, ob das Gerät benutzbar ist oder nicht.

Bei sequentiellen Programmen ist diese Technik durchaus brauchbar und nicht kritisch, anders bei parallel laufenden Prozessen.

Die einzelnen Flags, die eigentlich einzelne Bits, also logische Datenelemente sind, legt man in der Regel in einem Byte (bei Assemblerprogrammierung) oder als Boolesche Variable (in diversen Hochsprachen) ab, und fragt sie mit geeigneten Mitteln der jeweiligen Sprache auf ihren Zustand hin ab.

Soll diese Technik unter Multitask-Umgebung angewendet werden, so erscheint die folgende Lösung des Überwachungsrechnerproblems zunächst richtig. Die entsprechenden Befehle, hier in Z80-Assembler vorgestellt, dienen zum Belegen und Freigeben des Druckers. Die einzelnen Abschnitte sind in beiden Prozessen identisch. Das Flagbyte selber ist jedoch nur einmal im gemeinsamen Datenbereich vorhanden.

```
FLAGS   DB 0        ; Flagbyte im Datensegment, beiden
                    ; Prozessen zugaenglich.

; Prozesskode am Beispiel Prozess 1
;
                    ...
                    ; Belegen des Druckers, wenn nicht frei
Zeile:              ; dann warten.
1   WARTE1:  LD  A,(FLAGS)      ; Flagbyte holen und
2            BIT 0,A            ; nachsehen ob Bit0 gesetzt     Interrupt1
3            JP  NZ,WARTE1      ; wenn ja dann warten bis frei
4            SET 0,A            ; Drucker ist frei, also Belegen
5            LD  (FLAGS),A      ; durch Setzen des Bit0
                    ...
                    Benutzen des Druckers
                                                Interrupt 2
                                                (jedoch in Prozeß 2)
                    ...
                    LD  A,(FLAGS)       ; Freigeben des Bit0, da
                    RES 0,A             ; Drucker nicht mehr gebraucht
                    LD  (FLAGS),A       ; wird.
                    ...
```

Programm 3.1

Als Flagbyte wird hier ein Datenbyte mit dem Label FLAGS, dessen Bit Ø angibt, ob der Drucker von einem Prozeß gerade benutzt wird oder nicht, eingesetzt. Ist es auf Eins, so darf ein Prozeß, der drucken will, nicht fortfahren, sondern soll in einer Warteschleife auf die Freigabe warten.

Dies ist hier mit Z8Ø Mnemonics dargestellt. Durch einen BIT Befehl, der das angegebene Bit abprüft, gefolgt von einem bedingten Sprung, der sich auf das Bit bezieht, wird das Warten realisiert. Ist der Drucker frei, so belegt ihn der entsprechende Prozeß durch einen SET-Befehl, der das Signalbit in FLAGS setzt.

Danach benutzt ihn der jeweilige Prozeß, bis er seine komplette Ausgabe abgeschlossen hat. Will der andere Prozeß zu dieser Zeit ebenfalls den Drucker verwenden, so erkennt er anhand des BitØ in FLAGS, daß der Drucker bereits belegt ist und wartet in seiner Schleife, bis der erste Prozeß die Freigabe, durch das Rücksetzen des Bits anzeigt. Die Prozesse synchronisieren sich so aufeinander. Man spricht hier von Synchronisation, da sich die Prozesse untereinander verständigen können und sich dadurch aneinander orientieren.

Bei der Betrachtung der Flagtechnik muß man jedoch auch die Existenz des Systemkerns in seine Überlegungen mit einbeziehen.

Der Systemkern unterbricht zyklisch den gerade laufenden Prozeß und schaltet gegebenenfalls auf einen anderen um. Nehmen wir an, es sei hier das Round-Robin-Verfahren eingesetzt, so würde jeweils ein Zyklus an einem Prozeß gearbeitet und danach auf den anderen Prozeß umgeschaltet werden. Dabei könnte sich zufällig folgendes Zeitverhalten ergeben:

Prozeß 1 wird gestartet und läuft an. Weil er den Drucker verwenden will, gelangt er in den beschriebenen Programmteil und stellt dort fest, daß der Drucker frei ist. Deshalb würde er nicht warten, sondern weiterfahren. Soweit soll er jedoch nicht kommen, da nach Zeile 2 der Zeitinterrupt des Systemkerns zuschlägt (Programm 3.1). Demzufolge wird zunächst der Prozeß 2 fortgeführt. Auch er will in diesem Zyklus den Drucker verwenden und durchläuft demnach seine Zeilen 1 bis 5; damit kommt er in den Genuß des Druckers und verwendet ihn. Der nächste Zeit-Interrupt unterbricht ihn z. B. mitten in der geraden auszugebenden Zeile. Durch den Interrupt wird das Betriebssystem aktiviert, das Prozeß 1 fortführt und den Prozessor dazu auf Zeile 3 des Prozesses 1 aufsetzt.

Da die gesamten Registerstände, einschließlich dem Flagregister, wieder dem alten Stand entsprechen, wartet auch Prozeß 1 nicht, sondern schickt sich an, den Drucker zu verwenden. Dies ist jedoch fatal, da Prozeß 2 *noch nicht* mit seiner gesamten Ausgabe fertig ist. Damit ist die Unkorrektheit der Programmierung bewiesen.

Um dennoch das Problem mit Flags zufriedenstellend lösen zu können, müssen die entsprechenden Befehlssequenzen gegen Interrupts geschützt werden. Dies ist im Beispiel durch die Einrahmung der Abschnitte mit Disable Interrupt und Enable Interrupt möglich. Diese Befehle sperren bzw. geben die Interrupt-Funktion im Prozessor frei. Die Prozesse sehen dann korrekt so aus:

```
FLAGS   DB 0        ;Flagbyte im Datensegment, beiden
                    ;Prozessen zugaenglich.
; Prozesskode am Beispiel Prozess 1
;
            ...
            ; Belegen des Druckers, wenn nicht frei
            ; dann warten.

WARTE1:     EI                      ;Interrupt freigeben
            LD   A,(FLAGS)          ;Flagbyte holen und

            DI                      ;Interrupt sperren
            BIT  0,A                ;nachsehen ob Bit0 gesetzt
            JP   NZ,WARTE1          ;wenn ja, dann warten bis frei
            SET  0,A                ;Drucker ist frei, also Belegen
            LD   (FLAGS),A          ;durch Setzen des Bit0
            EI                      ;Interrupt freigeben
            ...


            Benutzen des Druckers
            DI                      ;Interrupt sperren
            LD   A,(FLAGS)          ;Freigeben des Bit0, da
            RES  0,A                ;Drucker nicht mehr gebraucht
            LD   (FLAGS),A          ;wird.
            EI                      ;Interrupt freigeben
            ...
```

Programm 3.2

Erst jetzt wären die Prozesse fehlerfrei. Die Programmsequenzen, die wie hier z. B. auf gemeinsame Daten zugreifen, werden als kritische Bereiche bezeichnet. Sie sind deshalb kritisch, da sie nicht ohne weiteres unterbrochen werden dürfen. Solche Bereiche sind daher in Disable Interrupt und Enable Interrupt zu klammern.

Dies gilt besonders auch für Systemprozeduren, die in der Regel, jede für sich, einen kritischen Bereich darstellen. Sie müssen direkt, nachdem sie aufgerufen werden, den Interrupt sperren und ihn erst unmittelbar vor der Rückkehr wieder freigeben.

Werden die Prozesse in einer nicht parallelen Hochsprache formuliert, so ändert sich an der Problematik nichts. Ein Compiler generiert aus einer Programmanweisung einen adäquaten Maschinen-Code, der dann zur Ausführung gebracht werden kann und sich zur Laufzeit nicht sonderlich von einem Assemblerprogramm unterscheidet.

Unser Problem könnte so formuliert werden:

```
var frei:boolean;
...
frei:=true;
...
(* Prozeß 1 *)
repeat until frei;                (* Warten bis Drucker frei ist *)
frei:=false;                      (* Drucker belegen *)
...
(* Benutzen des Druckers *)
...
frei:=true; (* Freigabe des Druckers *)
...
```

Programm 3.3

Da in einer Hochsprache das Schützen der kritischen Bereiche durch Disable – Enable Interrupt nicht ohne weiteres möglich ist, entspricht das Programm 3.3 der Lösung 3.1. Man müßte auch hier die Interrupts an entsprechender Stelle verbieten bzw. freigeben, was nur durch die Einbindung von Objektcode ins Pascal-Programm möglich ist.

Werden mehr als zwei Prozesse eingesetzt, so liegt der Fall nahe, daß einer den Drucker benutzt, die anderen hingegen in ihren Schleifen auf eine Freigabe warten. Wird er durch den benutzenden Prozeß freigegeben, so ist es rein zufällig, welcher Prozeß als nächster den Drucker belegt. Diese Zufälligkeit kann bei hohem Andrang auf den Drucker dazu führen, daß ein Prozeß nie oder erst nach sehr langem Warten den Drucker zur Verfügung gestellt bekommt.

Fassen wir die Eigenschaften der Methode zur Synchronisation von Prozessen mit Flags zusammen:

- Ein Prozeß, der auf einen anderen wartet, läuft in einer Schleife, die keine sinnvollen Tätigkeiten beinhaltet. Dadurch wird Prozessorzeit vergeudet, da der Prozeß auf ein Signal wartet, das frühestens nach dem nächsten Uhrinterrupt von einem anderen Prozeß aus gesetzt werden kann.
- Da mehrere Prozesse auf ein gemeinsames Datenwort (FLAGS) schreibend zugreifen, existieren kritische Codebereiche, die, um einen fehlerfreien Durchlauf zu gewährleisten, in Disable und Enable Interrupt eingerahmt werden müssen. Diese Befehle sind zum einen nicht überall verfügbar (Hochsprachen), und zum anderen haben sie große Tragweiten und sollten deshalb konsistenterweise in die Verantwortung des System-

kerns gelegt werden. Man denke an die Folgen, wenn ein Prozeß, durch einen Flüchtigkeitsfehler des Programmierers, fälschlich nie mehr den Interrupt freigeben würde.

- Das Setzen des Flags und damit die Belegung eines Betriebsmittels, geht zur Laufzeit mit dieser Methode sehr schnell. Nur wenige Befehle sind nötig, um die entsprechenden Aktionen auszuführen. Auch in einer Hochsprache geht das Setzen und Abfragen einer Booleschen Variablen relativ schnell.
- Warten mehrere Prozesse auf die Freigabe des Flags, so ist es nicht voraussehbar, welcher von ihnen als nächster das Bit setzt und sich dadurch die Verfügungsgewalt über das Betriebsmittel verschafft. Dadurch wird das Verhalten des Systems durch eine unerwünschte Zufälligkeit geprägt.

Wägt man jetzt die Vor- und Nachteile der Flagmethode gegeneinander ab, so muß man zu dem Schluß kommen, daß diese Methode, im Zusammenhang mit einem Multitasksystem, wenig bzw. nur unter speziellen Umständen brauchbar ist. Daraus resultiert der Bedarf nach einem besseren und stabileren Konzept, wie es von Semaphoren bereitgestellt wird.

3.1 Was ist eine Semaphore und wie setzt man sie ein

Das Semaphorkonzept wurde 1965 erstmalig von E. W. Dijkstra vorgestellt und dient zur Synchronisation paralleler Programme. Das Wort kommt ursprünglich aus dem Griechischen und bedeutet dort Signal.

Aus der Sicht der Informatik stellt eine Semaphore eine Datenstruktur dar, auf die grundsätzlich zwei Operationen anwendbar sind. Die Operationen werden durch zwei Prozeduren verkörpert, die ins Betriebssystem eingebettet sind. Mit ihnen ist es möglich, diverse Resourcen (Betriebsmittel) vor einem Doppelzugriff zu schützen. Als Bezeichnung für die beiden Operationen haben sich „wait" und „signal" eingebürgert. Sie sollen auch hier verwendet werden.

Durch die entsprechenden Aufrufe von „wait" bzw. „signal" wird auf eine Semaphorvariable zugegriffen.

Die Prozedur „wait" meldet, daß der aufrufende Prozeß die Semaphore, genauer gesagt das mit ihr verknüpfte Betriebsmittel, verwenden will. Da die Prozedur ein Teil des Systemkerns ist, entscheidet er, ob der Prozeß weiterlaufen und damit die Semaphore belegen darf, oder ob er verzögert werden muß. Dies ist dann zu tun, wenn schon ein anderer Prozeß die Semaphore durch einen „wait" Aufruf belegt hat.

Der Prozedur „wait" steht die Prozedur „signal" gegenüber. Sie erlaubt einem Prozeß, eine Semaphore wieder freizugeben. Durch einen Aufruf signalisiert man dem Systemkern, und über ihn den anderen Prozessen, daß z. B. ein bestimmtes Gerät wieder zur Disposition freisteht.

Das Problem der Konkurrenz um einen Drucker könnte wie folgt mit einer Semaphore gelöst werden. Anzumerken ist, daß bei Programm 3.4 nur eine einzige Semaphore existieren soll.

```
; Prozeßkode (für Prozeß 1 und 2 gleich)
  ...
CALL WAIT          ; Reservierung des Druckers
  ...
; Benutzen des Druckers
  ...
CALL SIGNAL        ; Freigabe des Druckers
  ...
```

Programm 3.4

Prozeß 1 hat Daten produziert und will diese drucken. Er ruft deshalb zunächst die Prozedure WAIT im Systemkern auf. Sie erkennt, daß die Semaphore frei ist und belegt sie durch den Prozeß 1. Er wird durch einen Return in den Prozeß fortgesetzt und kann den Drucker uneingeschränkt verwenden.

Will indessen Prozeß 2 auch den Drucker in Anspruch nehmen, ruft auch er zunächst ebenfalls die Prozedur WAIT auf. Da die Semaphore durch Prozeß 1 belegt ist, darf er nicht fortgeführt werden, sondern muß warten, bis die Sempahore freigegeben wird.

Die Verzögerung wird nicht durch eine Schleife, sondern durch einen Mechanismus im Betriebssystem bewerkstelligt. Bildlich gesprochen könnte man sagen, der Prozeß verschwindet im System und kommt erst wieder zurück, wenn Prozeß 1 durch den Aufruf von SIGNAL die Semaphore wieder freigibt.

Der SIGNAL Aufruf hat eine Doppelwirkung. Erstens meldet Prozeß 1 die Semaphore wieder frei und zweitens wird erkannt, daß ein weiterer Prozeß (Prozeß2) Bedarf angemeldet hat; durch ihn wird die Semaphore sofort wieder belegt und kommt so zur weiteren Ausführung.

Erweitert man das Beispiel noch um einige Tasks, so muß ein Weg gefunden werden, der die Fortführung wartender Prozesse regelt. Der Systemkern merkt sich die Reihenfolge, in der sich die Prozesse verzögerten und läßt bei einem SIGNAL-Aufruf den Prozeß fortfahren, der am Anfang der Reihe steht. Intern verfügt der Kern über eine Warteschlange, in die die Prozesse eingereiht werden, die sich durch eine WAIT Funktion verzögert haben.

Die Warteschlange ist in aller Regel nach dem Prinzip first in first out (FIFO) organisiert. Dies besagt, daß der Prozeß als erster ausgelöst wird, der auch als erster eingetreten ist.

Dieses Semaphorenkonzept sieht nun zunächst der Lösung mit dem Flag sehr ähnlich. Dabei unterscheidet sich die Logik des Belegens und Freigebens nicht sonderlich von der Flag-Methode, jedoch aber die inneren Funktionen der Prozeduren. Alle nachteiligen Eigenschaften der Flag-Methode sind bei Semaphoren nicht gegeben.

- Muß ein Prozeß warten, so wird dies vom Systemkern kontrolliert. Es wird nicht durch eine sinnlose Schleife, die nur dazu dient, Zeit zu schinden, erreicht. Wartet ein Prozeß, so wird dabei die nicht benötigte Zeit anderen Prozessen zur Verfügung gestellt; ähnlich der Prozedur DELAY aus Kapitel 2.
- Die Befehle, deren Benutzung nicht ungefährlich ist, wie Sperren von Interrupts, werden aus dem Benutzerbereich herausgenommen und gänzlich auf die Ebene des Betriebssystems verlegt. Dadurch wird das Gesamtsystem stabiler, und der Task-Programmierer braucht sich nicht um Details zu sorgen. Er kann die Prozeduren als Black Box benutzen.
- Die Auswahl des Prozesses, der fortgeführt werden soll, wird durch eine gerechte Warteschlangenstruktur unterstützt. Sie erfolgt nach dem FIFO Prinzip und ist nicht zufällig wie bei der Methode mit dem Flag.

Im vorigen Beispiel wurde nur eine Semaphorvariable, die keinen eigenen Namen hatte, benutzt. In realen Systemen existieren jedoch meistens mehrere Semaphoren, die dann durch eine Nummer oder einen Namen identifiziert werden. Die Nummer oder der Name ist beim Aufruf von „signal" bzw. „wait" in einer geeigneten Weise mit anzugeben.

An dieser Stelle soll zur weiteren Verdeutlichung ein Beispiel eingeführt werden, das auch in weiteren Kapiteln die jeweilige Thematik erklärt.

Es handelt sich um ein klassisches Teilnehmerproblem, nämlich um einen Mailbox-Rechner. Es ist ein System, in das sich Benutzer per Telefon einwählen und dann Nachrichten an spezielle Teilnehmer deponieren können. Ihrerseits haben sie die Möglichkeit, in ihren eigenen Fächern die für sie abgelegten Texte abzuholen. Ferner gibt es ein sog. Schwarzes Brett, auf das jeder schreiben und von dem jeder lesen kann. *Abbildung 3.1* zeigt die vorläufige Struktur der Software.

Das System soll physikalisch über vier Telefonanschlüsse verfügen; damit können vier Endbenutzer parallel bedient werden. Jedem Telefonanschluß ist eine USER Task zugeordnet, die den Dialog mit einem eingewählten Benutzer führt. Die USER Tasks sind untereinander, bis auf ihre Telefonnummern, gleich.

Das Schwarze Brett wird durch einen Arbeitsspeicherbereich realisiert, der für alle USER Tasks zugänglich sein muß. Der Zugriff auf das Schwarze Brett wird durch die Semaphore mit der Nummer 1 koordiniert. Die Struktur einer USER Task kann dann wie folgt aussehen:

```
; USER Task

  ...

WAIT(1)                ;Benutzung des Schwarzen Bretts anmelden
```

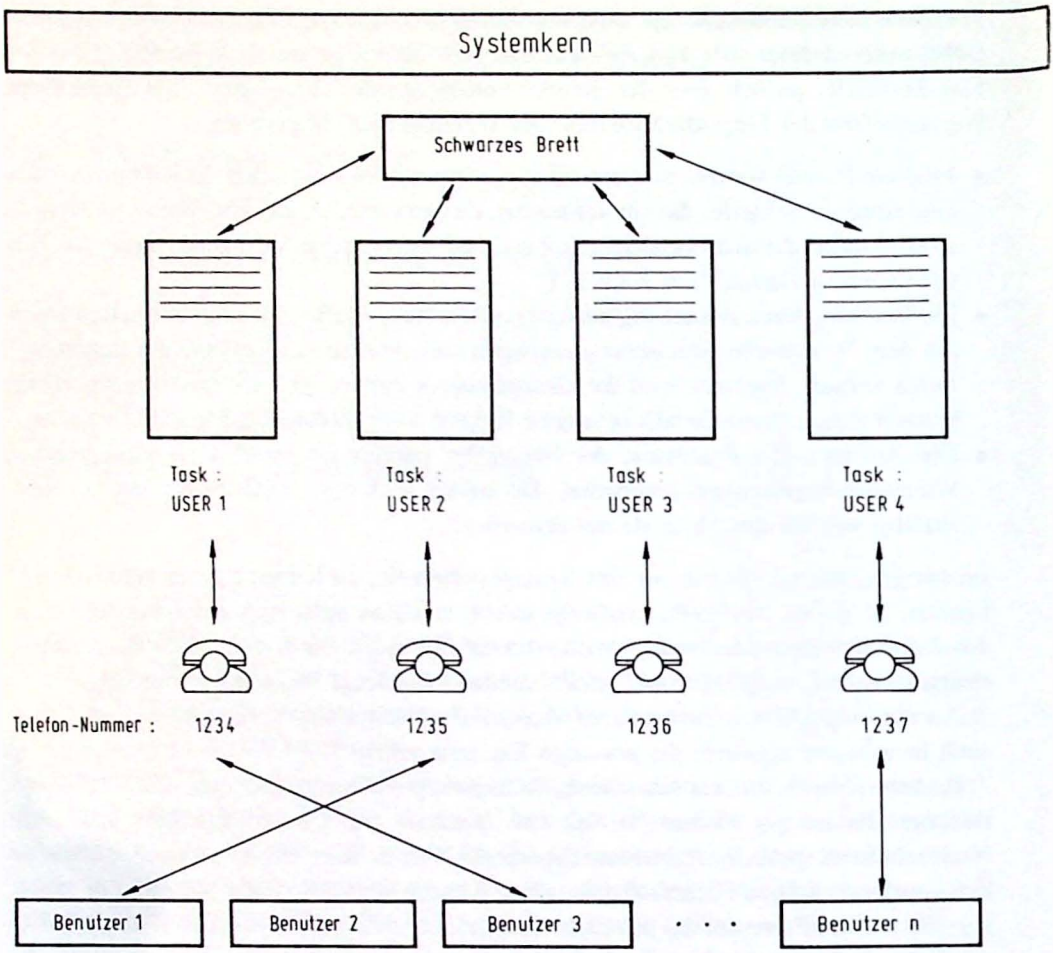


Abb. 3.1 Struktur des Mailbox Rechners

```
  ...
; Benutzen des Schwarzen Bretts
  ...
SIGNAL(1)              ;Freigeben der Benutzung
  ...
```

Mit Hilfe der Semaphore 1 wird vermieden, daß mehrere Benutzer gleichzeitig auf denselben Speicherbereich schreiben. Dadurch wird ein Chaos (Gemisch von Buchstaben) verhindert.

Anhand der genannten Beispiele und Argumente kann man festhalten, daß das Semaphor-Konzept eine saubere und klare Synchronisation von Prozessen, die gemeinsame Mittel benutzen, zuläßt. Das anschließende Unterkapitel zeigt auf, wie die Semaphoren im Betriebssystem realisiert werden können.

3.2 Realisierung von Semaphoren

Eine Semaphorvariable besteht aus zwei unterstrukturierten Elementen. Das erste Element ist ein Zähler, welcher ganze positive Zahlen einschließlich der Null beinhalten kann. Das zweite Element ist eine Warteschlange, in die sich beliebig viele Prozesse einhängen können. Die beiden Datenelemente bilden zusammen eine Semaphore.

Die Operationen, und damit die entsprechenden Systemprozeduren, sind vergleichsweise zur Datenstruktur komplizierter. Die Wirkung von „wait" und „signal" wurde zwar bereits besprochen, jedoch bedarf es zur Integration der Routinen in den Systemkern weiterer Ausführungen.

WAIT

Durch „wait" soll ein Prozeß gegegenenfalls verzögert werden, bis ein „signal" auftritt. Anhand der Zählervariablen entscheidet der Kern, ob eine Verzögerung nötig wird oder nicht. Er verzögert den aufrufenden Prozeß nur dann, wenn der Zähler gleich Null ist. In allen anderen Fällen wird der Zähler dekrementiert (um Eins vermindert) und der Prozeß fortgeführt; dies gilt auch dann, wenn der Zähler dadurch zu Null wurde.

Dem Zähler kommt eine besondere Bedeutung zu. Er zählt alle gesendeten und noch nicht empfangenen Signale. Also dient die Prozedur „wait" zum Empfangen und die Prozedur „signal" zum Senden von Signalen.

Die Notwendigkeit, daß Prozesse verzögert werden müssen, ergibt sich lediglich daraus, daß ein Prozeß, der ein Signal empfangen will, aber noch keines gesendet wurde, eben auf dieses warten muß.

Da der Zähler die gesendeten Signale zählt, kann der Kern feststellen, ob ein Prozeß weiterlaufen darf oder ob er zu verzögern ist. *Abbildung 3.2* zeigt den prinzipiellen Ablauf der „wait" Prozedur.

SIGNAL

Ein Prozeß ruft „signal" auf, wenn er das Auftreten eines bestimmten Umstandes (z. B. Drucker ist wieder frei) signalisieren will. Die Prozedur hat intern eine Verzweigung, deren

Abb. 3.2 Semaphoroperation WAIT

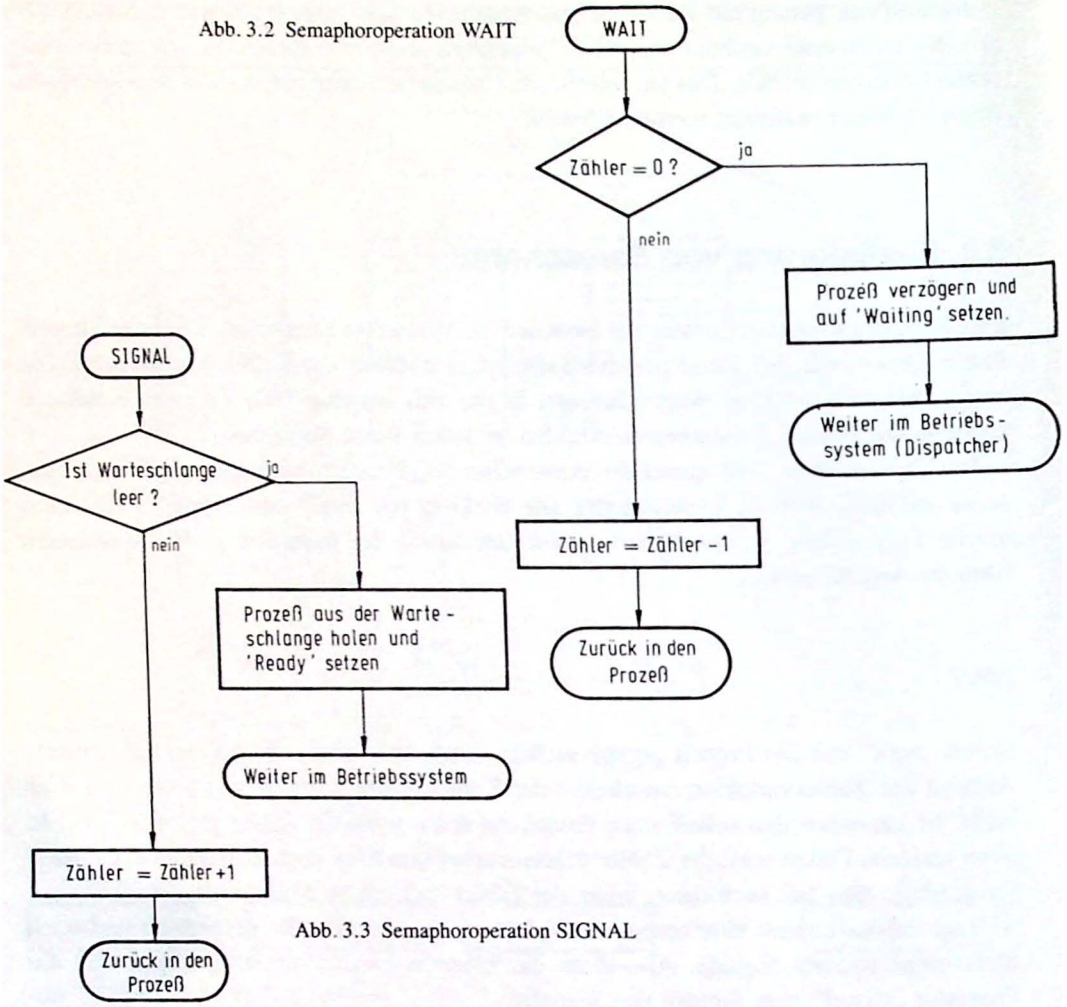


Abb. 3.3 Semaphoroperation SIGNAL

Kriterium die Warteschlange ist. Wartet kein Prozeß (Warteschlange ist leer), wird lediglich der Zähler um Eins inkrementiert (erhöht). Wird „signal" in diesem Zustand öfters aufgerufen, so hat dies nur eine jeweilige Erhöhung des Zähler zufolge.

Warten hingegen Prozesse in der Warteschlange, so wird der Zähler nicht manipuliert (er hat dann immer den Wert Null), sondern die Prozedur wählt den nächsten Prozeß aus der Warteschlange aus und setzt ihn fort *(siehe Abb. 3.3)*.

Bei der Grundinitialisierung (Urstart) des Betriebssystems müssen auch die Elemente der Semaphoren auf definierte Werte gebracht werden. Dabei wird die Warteschlangenkomponente generell auf leer und der Wert des Zählers auf Eins gesetzt. Dies müßte man nachträglich für Programm 3.4 voraussetzen, da sonst nicht der korrekte Ablauf erfolgt.

In manchen Systemen gibt es eine Prozedur, mit deren Hilfe der Programmierer eine Semaphore auf einen gezielten Wert initialisieren kann.

Aufbau der Warteschlangen

Um einen Prozeß warten zu lassen, sind Eingriffe in den Systemkern sowie in die Prozeßdeskriptoren nötig. Betrachtet man Abb. 2.10 mit dem Hintergedanken von Semaphoren, so erscheint kein Zustand für die Verzögerung eines Prozesses passend.

Der Zustand „delayed" ist ungeeignet, da er eine Verzögerung um eine bestimmte Zeit repräsentiert. Bei einer Semaphore ist jedoch die Dauer der Verzögerung im voraus nicht bekannt. Sie ist einzig davon abhängig, wann ein anderer Prozeß ein Signal sendet.

Deshalb muß ein neuer Zustand, der die Bezeichnung „waiting" trägt, eingeführt werden *(Abb. 3.4)*. Wird ein Prozeß, durch „wait" in die Warteschlange einer Semaphore gebracht, so trägt die Prozedur im Prozeßdeskriptor den neuen Zustand „waiting" ein. Wird dieser Prozeß durch einen „signal" Aufruf wieder aus der Warteschlange genommen, so geht sein Zustand von „waiting" nach „ready" über. Der Prozeß steht somit wieder zur Verarbeitung an. Erst der Dispatcher entscheidet später, welcher Prozeß als nächster den Prozessor erlangt.

Der Zustand „waiting" sagt allein jedoch noch nichts über die Reihenfolge, wie die Prozesse in diesen Zustand gelangten, aus. Die eigentliche Warteschlange wird über eine verkettete Liste von Prozeßdeskriptoren erreicht.

Jeder Deskriptor wird um einen Zeiger erweitert, mit dessen Hilfe, eine Kette von Deskriptoren gebildet werden kann. Als Aufhänger gilt eine Ankeradresse, die jeder Semaphore zu Eigen ist. Eine Semaphore besteht also endgültig aus dem Zähler und einem Zeiger, der, falls Prozesse warten, auf den entsprechenden Deskriptor weist.

Abbildung 3.5 zeigt eine Kette von drei verzögerten Prozessen, die auf ein Signal zur Semaphore 1 warten. Ein weiterer Aufruf von „wait", aus einem vierten Prozeß heraus, würde dazu führen, daß sein Prozeßdeskriptor in der Kette hinten angehängt wird.

Ruft ein anderer Prozeß „signal" auf Semaphore 1 auf, so soll der erste Prozeß, der in der Kette steht, wieder ausgehängt und fortgeführt werden. Deshalb setzt „signal" die Anker-

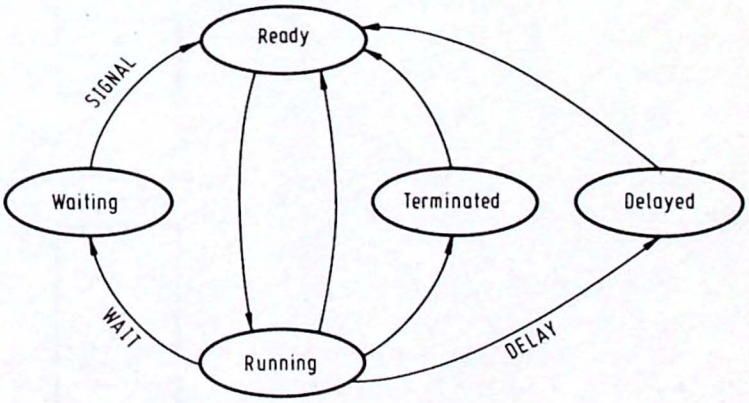


Abb. 3.4 Erweitertes Zustandsdiagramm durch Semaphoren

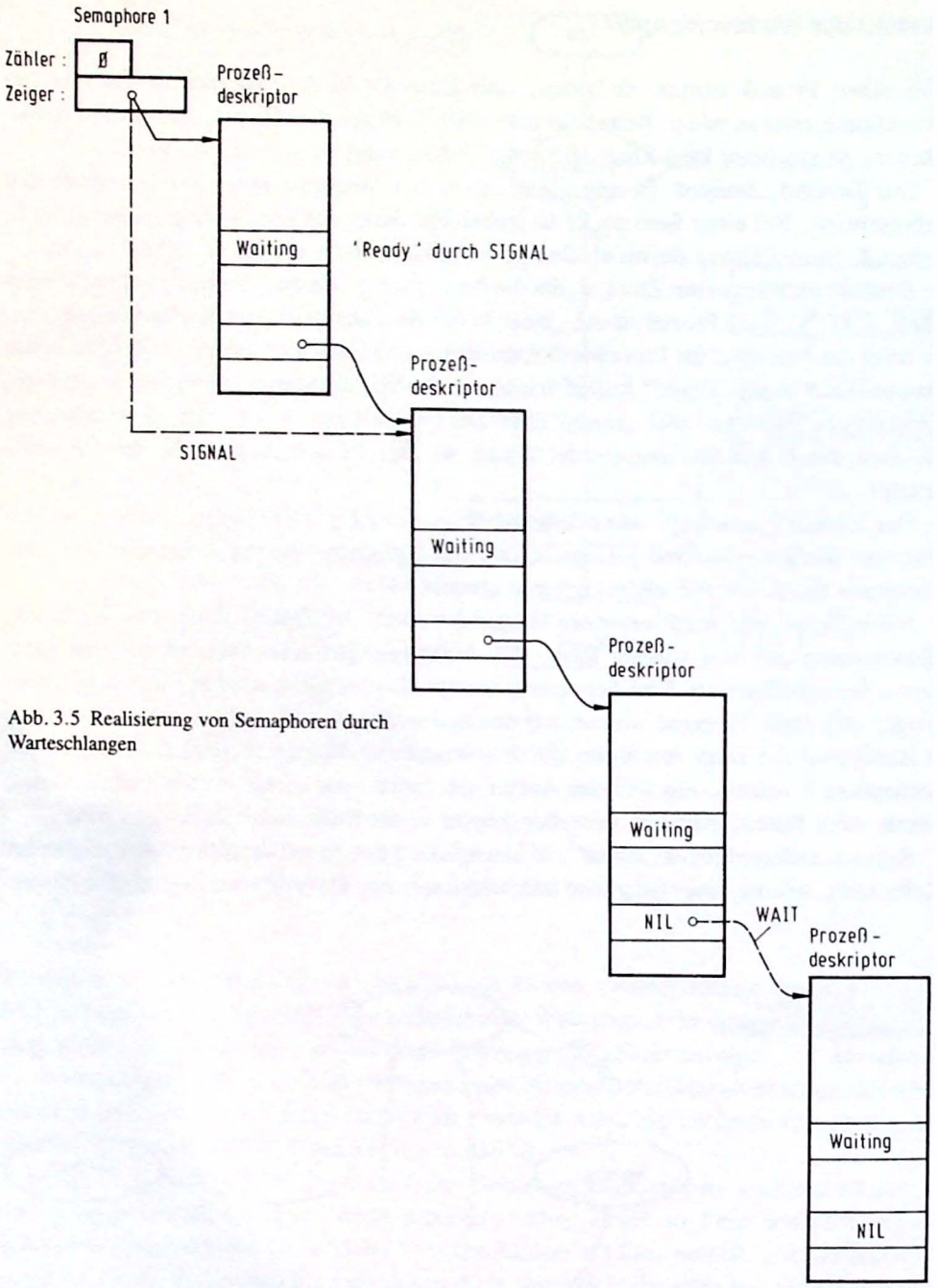


Abb. 3.5 Realisierung von Semaphoren durch Warteschlangen

adresse der Semaphore 1 auf den zweiten Prozeßdeskriptor um und ändert darüber hinaus den Zustand des ersten Prozesses von waiting nach ready.

Durch diese Zeigermethode läßt sich eine Warteschlange sehr schnell und einfach verwalten. Gibt es mehrere unterschiedliche Semaphoren, so könnte sich eine Struktur nach *Abbildung 3.6* ergeben.

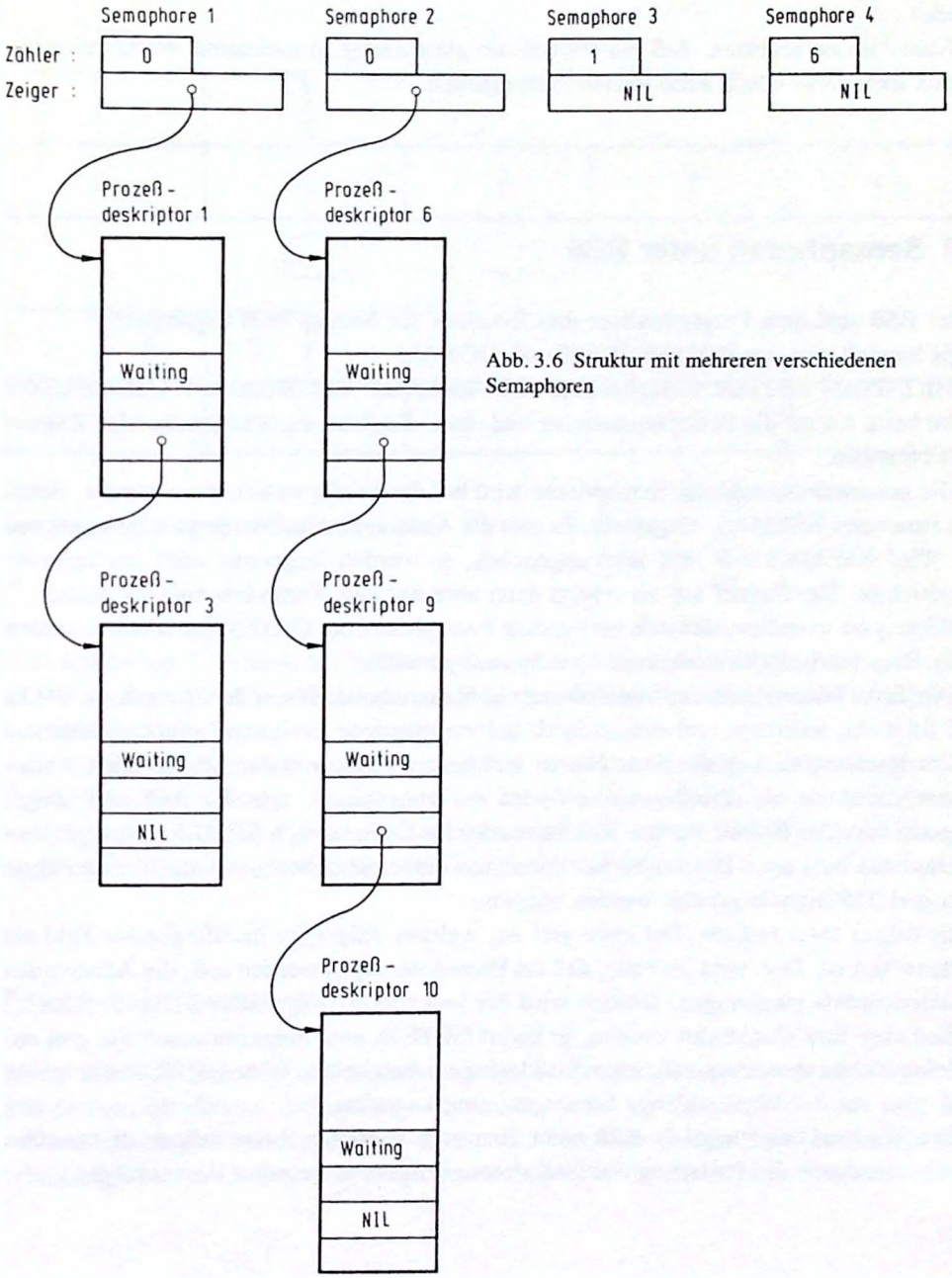


Abb. 3.6 Struktur mit mehreren verschiedenen Semaphoren

Hier sind vier Semaphoren, sowie eine Anzahl von Prozessen, präsent. Dabei sind Prozeß 1 und Prozeß 3 in der Semaphore 1 verzögert; Prozeß 6, 9 und 10 in Semaphore 2. Geht man davon aus, daß alle Semaphoren mit Eins initialisiert wurden, so kann man sagen, daß Semaphore 3 im Grundzustand ist. Auf Semaphore 4 hingegen sind fünf Signale gesendet worden, folglich steht der Zähler auf sechs. Dies wiederum hätte zur Folge, daß ein Prozeß erst nach der sechsten „wait" Operation auf Semaphore 4 verzögert werden würde!

Weiter ist zu beachten, daß ein Prozeß nie gleichzeitig in mehreren Warteschlangen warten darf. Dies würde auch keinen Sinn machen.

3.3 Semaphoren unter BSØ

Unter BSØ sind dem Programmierer drei Routinen für Semaphoren zugänglich.

Es handelt sich um INISEM, WAIT und SIGNAL.

Mit INISEM wird eine Semaphore auf einen konkreten Wert initialisiert. Dabei muß der Akku beim Aufruf die Semaphornummer und das C-Register den entsprechenden Zählerwert enthalten.

Die maximale Anzahl von Semaphoren wird bei der Konfiguration des Systems, durch den Parameter NSEMAS, eingestellt. Er gibt die Anzahl der einzurichtenden Semaphoren an. Wird NSEMAS z. B. mit zehn angegeben, so werden insgesamt zehn Semaphoren eingerichtet. Der Zugriff auf sie erfolgt dann aber mit den Nummern null bis neun!

Wichtig ist weiterhin, daß jede verwendete Semaphore über INISEM initialisiert werden muß. Dies wird nicht automatisch vom System gemacht.

Die Tasks können nach der Initialisierung in beschriebener Weise die Prozeduren WAIT und SIGNAL aufrufen, und sich dadurch auf verschiedene Ereignisse synchronisieren.

Die Implementierung der Semaphoren weicht hier etwas von der aufgezeigten Warteschlangemethode ab. Pro Semaphore wird ein Datenblock, wie ihn *Abb. 3.7* zeigt, eingerichtet. Die Blöcke werden hintereinander im Datenbereich SEMAS abgelegt.

Das erste Byte eines Blockes enthält den Zähler, dabei ist zu beachten, daß in einem Byte maximal 255 Signale gezählt werden können.

Es folgen zwei Indizes. Der erste gibt an, welcher Zeiger im nachfolgenden Feld als nächster frei ist. Dort wird im Falle, daß ein Prozeß verzögert werden soll, die Adresse des Taskdeskriptors eingetragen. Danach wird der Index PUTIDX erhöht.

Soll eine Task fortgeführt werden, so liefert GETIDX eine Zeigernummer. Sie gibt an, welcher Zeiger den entsprechenden Taskdeskriptor beinhaltet. Über GETIDX kann eine Task also aus der Warteschlange herausgenommen werden.

Die Warteschlange wird in BSØ nicht über eine verkettete Liste aufgebaut, sondern ergibt sich durch die Pufferung der Deskriptorenadresse in richtiger Reihenfolge.

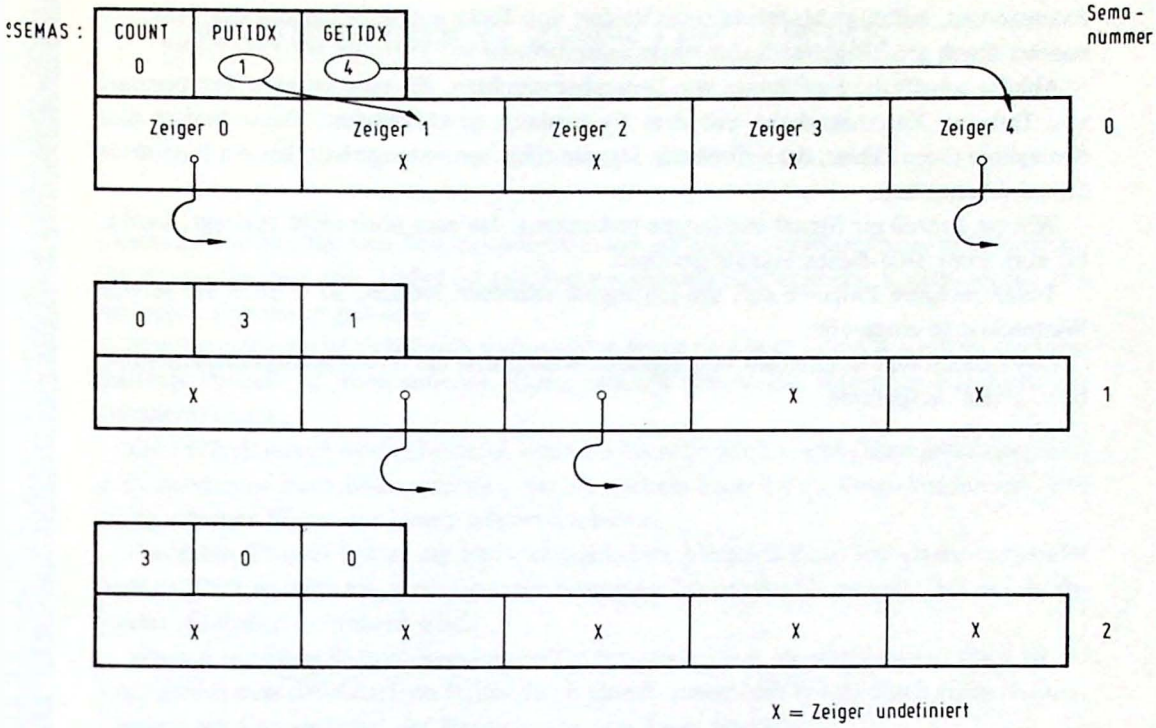


Abb. 3.7 Realisierung von Semaphoren im BSØ

In Abbildung 3.7 würde bei der Semaphore 1 der Index auf den Warteschlangenbeginn auf Zeiger 1 weisen. Ein anzuhängender Deskriptorblock würde bei Position 3 eingesetzt werden.

Es sind pro Semaphore soviel Zeigerplätze im Feld eingerichtet, wie die maximale Anzahl von Tasks beträgt. Ferner muß bei der Verwaltung der Indizes darauf geachtet werden, daß am Ende des Feldes ein Rücksetzen auf Null erfolgt. Dies wird mit Hilfe einer Funktion *MOD*, die den *Rest* einer Division liefert, erledigt.

Zusammenfassung

Um parallele Programme untereinander sauber kooperieren zu lassen, ist eine Methode erforderlich, die diesem Anspruch genügt.

Die Methode der Steuerung durch Flags erscheint nicht geeignet, da sie zwar formal richtig arbeitet, jedoch aber nachteilige Eigenschaften besitzt. Diese sind: Vergeudung von

Prozessorzeit, zufällige Momente beim Restart von Tasks sowie Belastung des Programmierers durch grundlegende Synchronisationsprobleme.

Abhilfe schafft die Einführung von Semaphorvariablen, die eine saubere Verzögerung von Tasks in Zusammenhang mit dem Systemkern gewährleisten. Dazu besitzt eine Semaphore einen Zähler, der auftretende Signale zählt und so lange hält, bis ein Prozeß ein Signal erkannt hat.

Will ein Prozeß ein Signal empfangen (erkennen), das aber noch nicht vorliegt, wird er bis zum Eintreffen dieses Signals gestoppt.

Treten mehrere Prozesse auf, die ein Signal erkennen wollen, so werden sie in eine Warteschlange eingereiht.

Das Senden und Empfangen von Signalen wird durch die Systemprozeduren „signal" bzw. „wait" ausgeführt.

4 Datentransport zwischen Tasks

Durch die Einführung von Semaphoren ist es uns gelungen, parallele Prozesse untereinander zu synchronisieren. Dabei ist mit Synchronisation eine zeitliche Koordinierung von mehreren Prozessen gemeint.

Mit Semaphoren ist es bis jetzt noch nicht möglich gewesen, Daten von einem zu einem anderen Prozeß zu transportieren. Dazu nämlich dienen die folgenden Konzepte zur Datenpufferung.

Ein Pufferkonzept wird gebraucht, wenn ein Prozeß irgendwelche Daten produziert, also z. B. durch eine Berechnung erzeugt, und ein weiterer Prozeß diese Daten konsumiert, also in irgendeiner Weise die Daten weiterverarbeitet.

Weiterhin können Puffer zur zwischenzeitlichen Abspeicherung von Daten verwendet werden. Dies ist dann relevant, wenn ein erzeugtes Datenmaterial erst später bei weiterführenden Aktionen gebraucht wird.

Werden in einem System mehrere Puffer verwendet, was im allgemeinen üblich ist, so identifiziert man die einzelnen Puffer durch eine *Nummer* oder besser durch einen *Namen*. Ähnlich wird es auch bei der Bezeichnung von Tasks gemacht.

Die Pufferverwaltung eines Mehrprozeßsystems muß, um einen sicheren Ablauf zu gewährleisten, folgenden grundlegenden Anforderungen genügen:

- Die zu transportierende Information darf in keiner Weise durch den Transport verändert werden.
- Es muß verhindert werden, daß ein datenproduzierender Prozeß auf einen bereits gefüllten Puffer, dessen Kapazität ausgeschöpft ist, schreibt.

Die Pufferverwaltung kann auf den zweiten Punkt dadurch reagieren, daß sie den Prozeß verzögert, bis ein Platz im Puffer frei wird. Eine andere Möglichkeit besteht darin, dem Produzenten durch eine Rückmeldung bekannt zu machen, daß die Information nicht abgelegt werden konnte. Dann ist der Prozeß jedoch für die weitere Verwendung seines Produktes selber verantwortlich.

- Ein Prozeß, der eine Nachricht aus einem Puffer nehmen will, dort jedoch keine findet, muß in geeigneter Weise gesteuert werden. Entweder, man *verzögert* ihn bis zum Eintreffen einer Information, oder man meldet dem Prozeß das Fehlschlagen der Leseoperation. Der Prozeß kann dann seinerseits auf diesen Fall, in einer vom Programmierer bestimmten Weise, reagieren.
- Es darf nie gleichzeitig (d. h. mehrere Tasks auf einmal) auf ein Pufferelement zugegriffen werden.

Da der Prozeßlauf durch die Pufferverwaltung beeinflußt wird (z. B. Verzögerung bis Puffer frei ist) und da es sich um grundlegende Dienste handelt, steht die Pufferverwaltung logisch in der Nähe des Systemkerns, und wird demnach in diesen integriert.

Die Realisierung der Verzögerungen könnte direkt in den Kern eingebracht werden. Dazu müßte man einen neuen Prozeßstatus sowie weitere Warteschlangen einführen.

Einfacher geht es, wenn man auf bereits eingeführte Methoden zurückgreift. Eine Pufferverwaltung kann unter Zuhilfenahme von Semaphoren aufgebaut werden.

4.1 Einfacher Datenpuffer

Greifen wir zur Einführung das Mailbox-Beispiel auf. Dabei ergänzen wir es durch eine weitere Task (CONSOLE), welche die Verbindung zu einem Konsoloperator herstellt.

Bei dem Mailbox-System soll ein eingewählter Benutzer von seiner USER Task aus eine Nachricht direkt an den Konsoloperator senden können. Die Information soll maximal 80 Zeichen lang sein. Die Task CONSOLE soll zunächst nur Nachrichten empfangen und nach Erhalt einer Meldung diese auf dem Bildschirm des Mailbox-Rechners ausgeben. Danach wartet sie auf die nächste Nachricht. *Abbildung 4.1* zeigt das ergänzte System.

Um eine Pufferverwaltung, die den Anforderungen aus Punkt 4 gerecht wird, aufzubauen, wird das System um den Puffer sowie um zwei Prozeduren „send" und „receive" erweitert. Dazu werden zwei Semaphorvariablen, die die anfallenden Verzögerungen sowie Taskfortsetzungen abwickeln, gebraucht. Programm 4.1 zeigt die Grundstruktur der Pufferverwaltung. Die Beschreibung der Programme erfolgt durch einen Pseudocode.

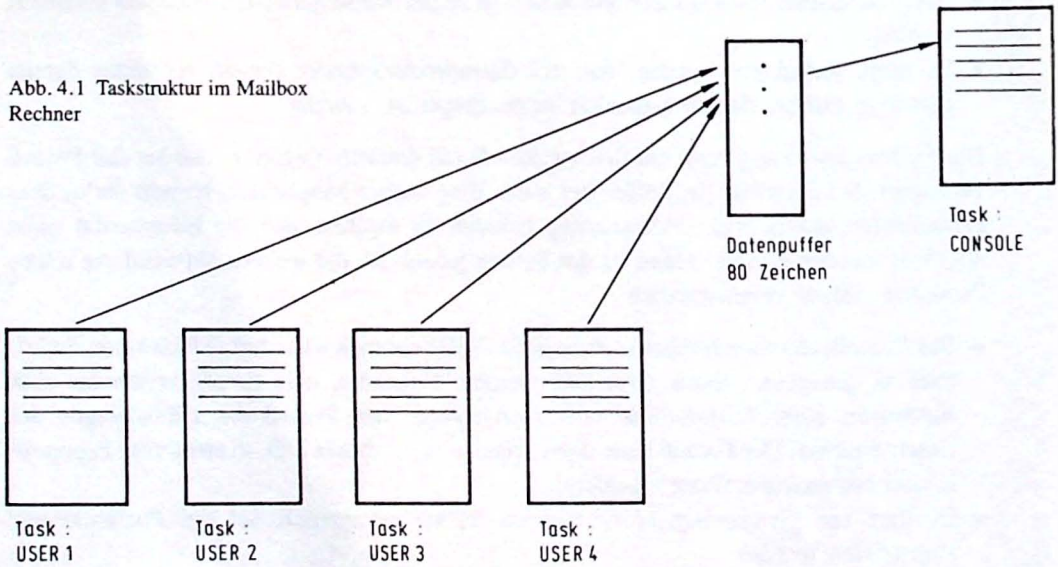


Abb. 4.1 Taskstruktur im Mailbox Rechner

```
prozedur send(...)
wait(1)
...
Schreiben in den Puffer
...
signal(2)
end

prozedur receive(...)
wait(2)
...
Lesen aus dem Puffer
...
signal(1)
end

Initialisieren der Semaphoren
init(1,1)   Semaphore 1 auf Eins setzen
init(2,0)   Semaphore 2 auf Null setzen

ask USER1
...
send(text)   Senden eines Textes
...
end

Task USER4
...
send(text)   Senden eines Textes
...
end

Task CONSOLE
...
receive(text)
write(text)
...
end
```

Programm 4.1

Im Programm 4.1 kommt es hier nicht darauf an, in welcher Weise die Parameter an die Prozeduren (send und receive) übergeben werden, vielmehr soll es die Synchronisation der Tasks aufzeigen.

Bei den Semaphoreprozeduren ist in Klammern die Nummer der Semaphore, auf die sich die Operation beziehen soll, angegeben. Die Pufferverwaltung verwendet somit die Semaphore Nummer 1 und 2.

Die gezielte Initialisierung der Semaphoren ist hier von großer Bedeutung. Semaphore 1, die signalisiert, ob der Puffer frei ist, muß auf Eins vorbelegt werden, da zu Beginn der Puffer immer leer ist. Ruft eine USER Task die Prozedur „send" auf, so wird zunächst ein „wait" auf Semaphore 1 ausgeführt, um sich den Zugriff auf den Puffer zu verschaffen. Senden mehrere USER nacheinander, so werden die Prozesse in die Warteschlange von Semaphore 1 eingereiht. Erst durch den Aufruf von „receive", durch den ein „signal" auf die Semaphore 1 ausgeführt wird, werden die wartenden Sender nacheinander fortgesetzt.

Semaphore 2 indessen wird zur Ablaufsteuerung von Empfängerprozessen (hier ist nur einer, nämlich CONSOLE vorhanden), gebraucht. Durch sie würden sich Prozesse verzögern, die auf eine Sendung warten, obwohl noch keine abgesetzt wurde. Diese Semaphore muß auf Null initialisiert werden, denn zu Beginn liegt noch kein Puffereintrag vor.

Durch den Gebrauch von Semaphoren ist eine effiziente und einfache Pufferverwaltung ermöglicht worden. Dabei können die Prozeduren „wait" und „signal" als Black Box betrachtet werden – wie eine Taskverzögerung im Detail abläuft, interessiert auf dieser Ebene nicht mehr. Durch das Zusammenspiel der Semaphoren wurde erreicht, daß ein Prozeß Daten erlangt, die asynchron zu seinem Verlauf erzeugt wurden.

4.2 „No Wait" Prozeduren

Bei einigen Problemen ist es durchaus sinnvoller, nicht auf eine Sendung zu warten. Dies träfe auch für das Beispiel der CONSOLE Task zu, da sie ständig auf den Empfang einer Nachricht wartet.

Besser wäre eine Prozedur „receive-no-wait", die auch dann zurückkehrt, wenn keine Information im Puffer steht. Dies muß sie jedoch der aufrufenden Task mitteilen, die danach entsprechend fortfährt. Die Task CONSOLE könnte so noch weitere Dienste übernehmen, z. B. den Dialog mit dem Operator führen.

Die Prozedur „receive-no-wait" hat folgende Struktur:

```
prozedur receive_no_wait(...)
...
if Puffer voll then
        ...
        wait(2)
        Lesen der Information aus dem Puffer
        Rückmeldung:= ok
        signal(1)
```

```
            . . .
            else Rückmeldung von Fehler
. . .
```

Programm 4.2

Demgegenüber kann auch eine „send-no-wait" Prozedur brauchbar sein. Auch sie soll sofort in die aufrufende Task zurückkehren; ungeachtet dessen, ob der Puffer voll ist oder nicht. Ob das Senden erfolgreich war wird der Task durch einen Parameter angezeigt.

```
prozedur send_no_wait(. . .)
. . .
if Puffer leer then
            . . .
            wait(1)
            Schreiben der Information in den Puffer
            Rückmeldung:= ok
            signal(2)
            . . .
            else Rückmeldung von Fehler
. . .
```

Programm 4.3

In Prozedur 4.2 und 4.3 gibt jeweils „Rückmeldung" der aufrufenden Task an, ob das Empfangen bzw. Senden erfolgreich war.

Neben den „send" und „receive" Prozeduren sollten noch zwei wichtige Funktionen realisiert werden.

Es handelt sich um „empty" und um „full"; dabei gibt „full" an, ob der Puffer voll ist und „empty" gibt an, ob er leer ist.

Die „no wait" Prozeduren reduzieren sich dann, wenn man „receive" und „send" voraussetzt, auf:

```
prozedur receive_no_wait(. . .)
. . .
if full then receive
            Rückmeldung:= ok
            else Rückmeldung von Fehler
. . .

prozedur send_no_wait(. . .)
```

```
...
if empty then send
            Rückmeldung:= ok

            else Rückmeldung von Fehler

...
```

Programm 4.4

Anmerkung: „full" und „empty" sind bei nur einem Pufferelement, wie es hier der Fall ist, gegenläufig, d. h.

empty = not full

Dies ändert sich jedoch im anschließenden Teilkapitel, wo mehrere Elemente im Puffer stehen werden.

4.3 Ringpuffer

Das in 4.1 erläuterte Pufferkonzept ist zwar brauchbar, es läßt aber nur das Zwischenspeichern eines Elementes zu. Im Mailbox-Beispiel besteht ein solches Element aus 80 Zeichen, also einer Zeile.

Die nachteilige Auswirkung auf den Gesamtablauf besteht darin, daß gegebenenfalls mehrere Sender verzögert werden müssen. Daher wird ein Puffer gebraucht, der mehrere Elemente halten kann, und eine Task nur dann verzögert, wenn alle vorgesehenen Plätze belegt sind.

Die sogenannten Ringpuffer bieten eine praktikable Lösung an. Im Pufferbereich werden Plätze für mehrere Elemente vorgesehen und über die „send" und „receive" Prozeduren als Ring organisiert. *Abbildung 4.2* zeigt einen Ringpuffer für maximal 8 Elemente, von denen vier belegt sind. Dabei kann es durchaus sein, daß eine Task mehrere Elemente abgesetzt hat. Dies trifft hier für Task 1 zu. Die Prozeduren „send" und „receive" verwalten die Pufferplätze. Soll eine weitere Information eingeschrieben werden, so schreibt „send" diese an den nächsten freien Platz. Im Beispiel ist es Platz 5, welcher durch eine Information von Task 2 belegt wird. Danach schaltet „send" auf den nächsten freien Platz, hier Platz 6.

Die Prozedur „receive" holt eine Information aus dem Puffer, indem sie ihr jeweiliges Element liest und auf das nächste umschaltet. Im Beispiel wird Element 1 gelesen und auf Element 2 fortgeschaltet. Dadurch wird der Platz 1 frei und kann durch „send" wieder neu belegt werden. Die Prozeduren „send" und „receive" rotieren dadurch ständig um den Puffer.

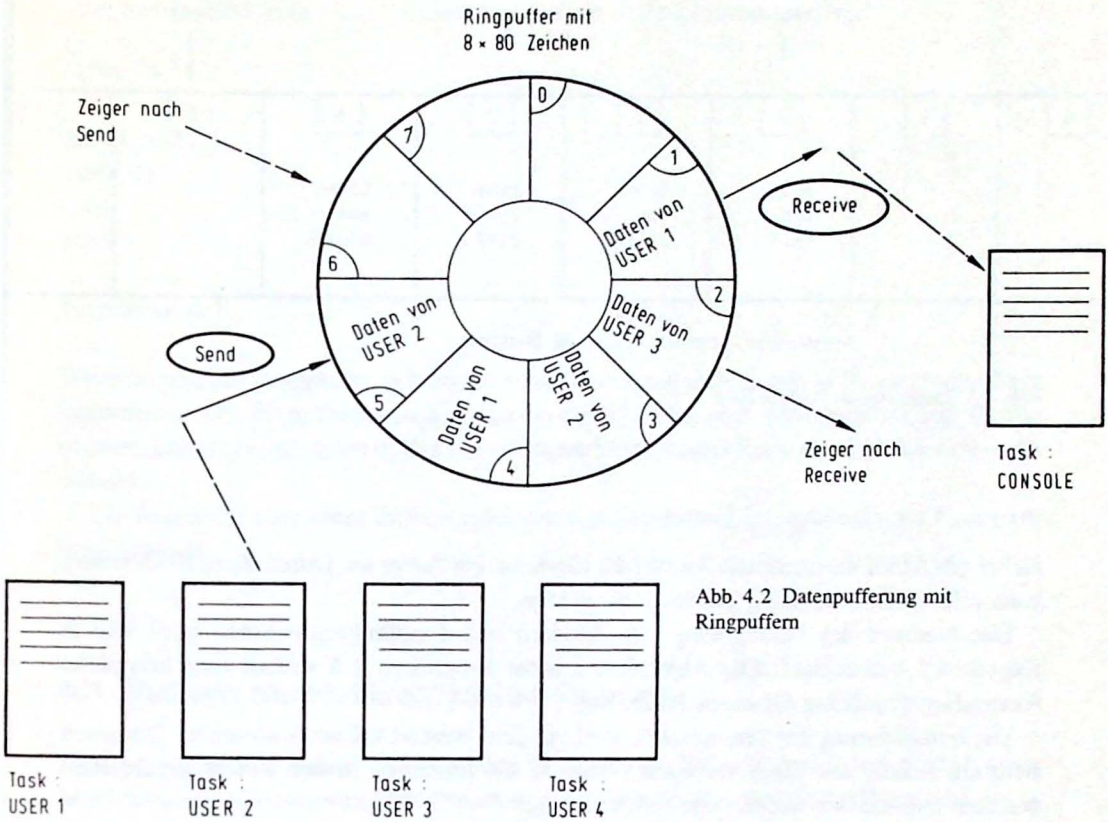


Abb. 4.2 Datenpufferung mit Ringpuffern

Da es im Rechner in der Regel keinen Ringspeicher gibt, wird der Ring durch ein lineares Datenfeld realisiert. Hierzu wird das eigentliche Datenfeld, sowie zwei Indizes zur Verwaltung gebraucht. Nach den oben beschriebenen Aktionen ergibt sich dann eine Struktur nach *Abb. 4.3*, die den Ringpuffer als lineares Datenfeld zeigt. Die beiden Indizes sollen C für „receive" und P für „send" heißen.

Im Beispiel steht C auf zwei und P auf sechs. Nach einer „send" oder „receive" Aktion muß der entsprechende Index inkrementiert werden. Dabei ist zu beachten, daß der Index beim Inkrementieren von 7 auf das nächste Element, nicht auf 8, sondern auf 0 zurückzusetzen ist. Das wird durch folgende einfache Berechnung am Beispiel des Index P erreicht:

P := (P+1) MOD 8

Die Operation MOD (modulo) liefert den Rest der Division von (P+1) durch 8. Dieser ist dann Null, wenn P+1 gleich 8 ist. Dasselbe gilt für den Index C. Bindet man sich nicht an 8 Elemente, so kann man die Formel allgemein definieren.

P := (P+1) MOD MAX

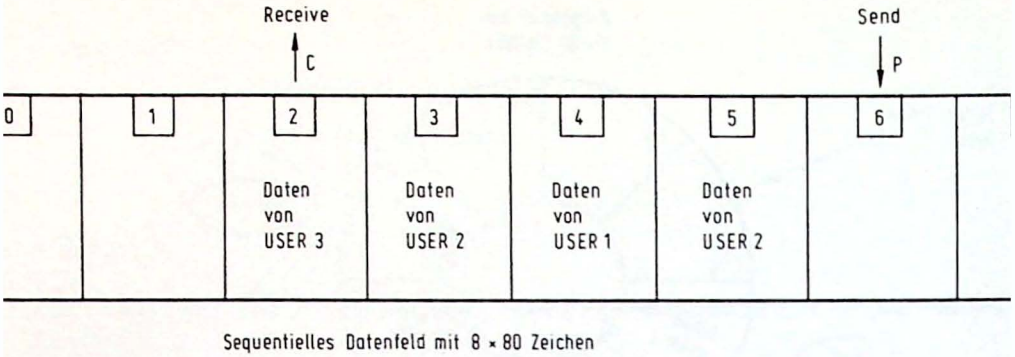


Abb. 4.3 Ringpuffer als lineares Feld

Dabei gibt MAX die maximale Anzahl der Elemente im Puffer an. Durch diese Berechnung kann sehr einfach der Ring geschlossen werden.

Das Konzept der Verzögerung von Sendern und Empfängerprozessen wird wie in Kapitel 4.1 beibehalten. Die Algorithmen unter Programm 4.5 stellen eine komplette Ringpufferverwaltung für einen Puffer vor.

Die Initialisierung der Semaphore 1 wird mit dem Wert MAX vorgenommen. Demnach wird ein Sender erst dann verzögert, wenn MAX Elemente in den Puffer geschrieben wurden. Die Indizes werden mit Null vorbelegt.

```
max = 8
puffer: Feld mit max Elementen

Prozedur send
...
wait(1)
...
puffer(p) := Information
p := (p+1) mod max          /auf das nächste leere Element schalten/
...
signal(2)

Prozedur receive
...
wait(2)
...
Information := PUFFER(c)
```

```
c := (c+1) mod max          /auf das nächste volle Element schalten/
...
signal(1)

init (1,max)
init(2,0)
c:=0
p:=0
```

Programm 4.5

Werden mehrere Ringpuffer gebraucht, so müssen diese jeweils eigene Semaphoren-Paare bekommen. Die Prozeduren „send" und „receive" sollen aber nur einmal existieren. Sie müssen jedoch gewährleisten, daß die richtigen Semaphoren-Paare und Indizes verwendet werden.

Die Indentifikation eines Puffers erfolgt wie bei Tasks und Semaphoren über Nummern bzw. Namen.

4.4 Puffern durch Nachrichten-Queues

Eine grundsätzlich andere Methode der Pufferung ergibt sich durch den Aufbau einer Warteschlange (Queue) von Datenelementen. Sie funktioniert ähnlich der Warteschlangenverwaltung bei Semaphoren, mit dem Unterschied, daß hier nicht Prozeßdeskriptoren verkettet werden, sondern Datenelemente.

Im Gegensatz zum Ringpuffer verwendet man hier kein starres Datenfeld zur Pufferung, sondern beschafft sich den benötigten Pufferbereich von einer dynamischen Speicherverwaltung (siehe Kapitel 7) oder von der aufrufenden Task.

Ist letzteres gegeben (ohne dynamische Speicherverwaltung) so erhält die Prozedur „send" lediglich einen Pointer auf die Information. Die Daten selber stehen im Datenbereich der Task. Das heißt, ein Sender stellt eine Variable aus seinem Datensegment zunächst der Pufferverwaltung und später einer empfangenden Task zur Verfügung!

Da die Information nicht echt gepuffert wird, muß der Sender dafür Sorge tragen, daß er die versendete Variable erst dann wieder verwendet, wenn sie vom Empfänger verarbeitet wurde. Dies kann dadurch erreicht werden, daß der Empfänger eine wesentlich höhere Priorität zugeteilt bekommt.

Aus Gründen der Sicherheit ist hier zu empfehlen, den anderen Weg, der über eine Speicherverwaltung, die aus einem Datenbereich (pool) Datenelemente dynamisch vergeben und wieder zurücknehmen kann, besteht, einzuschlagen.

In Abbildung 4.4 wird eine Nachrichtenschlange gezeigt. Sie besteht eigentlich aus nur einem Ankerzeiger, der auf das Element zeigt, welches als erstes ausgelöst werden soll.

Die einzelnen Elemente können hier verschiedene Längen haben, was bei Ringpuffern nicht möglich war. Das Ende kann z. B. durch den Stern angezeigt werden. Jedes Datenelement trägt einen Zeiger, der zum nächsten Element weist. Das letzte Element hat an dieser Stelle den Wert NIL. NIL bedeutet, daß kein weiteres Glied folgt.

Soll eine „send" Funktion auf die Warteschlange erfolgen, so wird zunächst das Ende gesucht (Element mit NIL) und dort das neue angehängt. Die Prozedur „receive" dagegen gliedert das erste Element aus, indem sie die Ankeradresse auf das zweite umsetzt.

Da eine solche Warteschlange theoretisch unendlich lang sein kann und damit nie voll wird, fällt die Verzögerung eines Senders zunächst weg. Die Verzögerung eines Empfängers, der auf eine leere Warteschlange (Anker=NIL) zugreift, kann jedoch sinnvoll sein.

Arbeitet man mit einer Speicherverwaltung, so muß die Prozedur „send" sich zunächst einen Speicherbereich beschaffen und kann dann erst die Originaldaten der Task dorthin kopieren.

Ist im gemeinsamen Speicher (pool) kein Platz mehr, so sollte sich der Sender durch eine Semaphore verzögern. Die „receive" Prozedur gibt nach dem Umkopieren des Elementes in eine Variable des Empfängers den Speicher an die Speicherverwaltung zurück, die diesen Bereich dann weiter vergibt. Dadurch könnte auch ein eventuell wartender Sender, der keinen Platz im gemeinsamen Speicher bekommen hat, wieder fortgeführt werden. Global beschrieben werden die „send" und „receive" Prozeduren durch Programm 4.6.

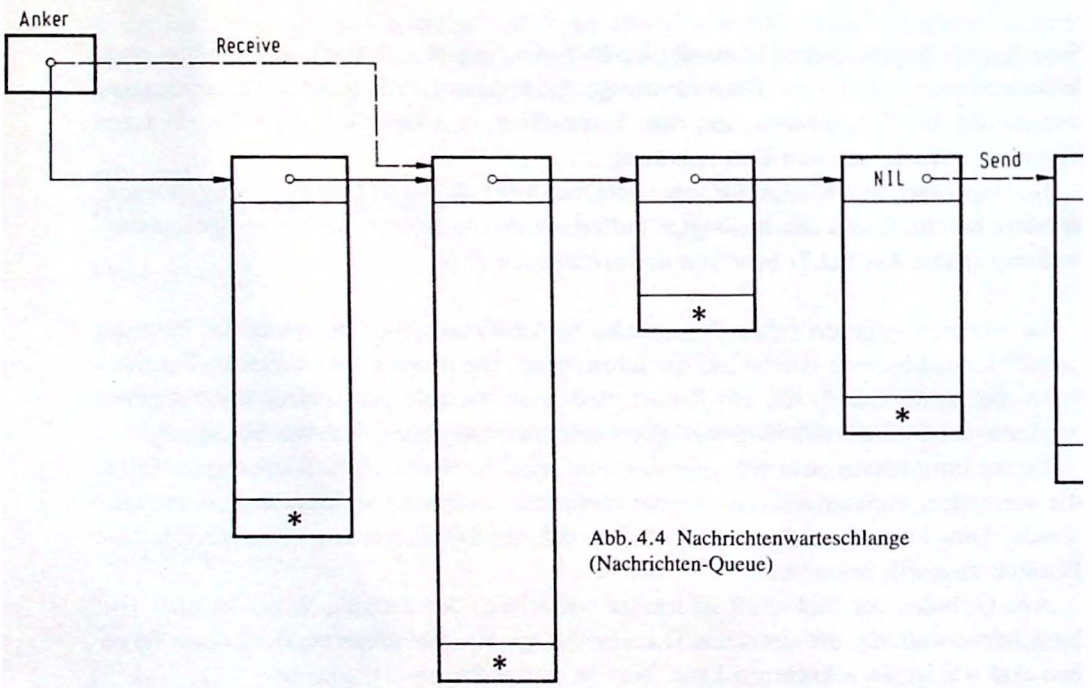


Abb. 4.4 Nachrichtenwarteschlange (Nachrichten-Queue)

Prozedur send(info)

...

Beschaffe Speicher aus dem Pool, wenn keiner vorhanden, dann verzögere Task.

...

Kopiere die Information in den bereitgestellten Speicher. Suche das Ende der Queue und setze dort im letzten Element den Zeiger auf das bereitgestellte Element. Setze Zeiger im letzten Element auf NIL.

...

Prozedur receive(info)

...

Speichere Daten wo Ankeradresse hinweist nach Variable
info.
Gib Speicher im Pool frei und führe eventuell verzögerte Task fort.
Setze Anker auf nächstes Element um.

...

Programm 4.6

4.5 Priorisierte Pufferverwaltung – Pufferstrategien

Bei allen bisher aufgezeigten Puffermethoden wurde konsistent eine first-in-first-out (FIFO) Strategie zum Auslösen eines Elementes aus einem Puffer verwendet. Das ist in den meisten Anwendungen am sinnvollsten; deshalb wird FIFO in der Praxis durchweg eingesetzt.

Es gibt aber daneben Problemstellungen, bei denen eine FIFO Pufferung nicht brauchbar ist. Werden Daten gepuffert, deren Bedeutung um so interessanter ist, je neuer sie sind; bietet sich die Verwaltung nach last-in-first-out (LIFO) an. Dort wird immer das zuletzt eingetragene Element an eine empfangende Task weitergegeben.

In Systemen, in denen Meldungen sporadisch anfallen, z. B. bei Überwachungs- und Steuerrechnern, müssen diese Meldungen in geeigneter Weise, bis sie verarbeitet bzw. quittiert sind, gepuffert werden. Hier kann das LIFO Prinzip hilfreich sein, indem z. B. die aktuellste Meldung angezeigt und vorhergehende Meldungen jeweils gepuffert bleiben. Wird die aktuellste Meldung quittiert, so erscheint die davor aufgetretene usw.

Man kann sich eine jeweilig passende Methode der Auslösungsreihenfolge aus dem Puffer ausdenken. Sie kann auch davon abhängig gemacht werden, inwieweit das eingetragene Element das System belastet. Es wäre denkbar, die Reihenfolge der Auslösung davon abzuleiten, wieviel ein Element an Speicher benötigt. So wäre dann das jeweils größte Element eines Puffers beim Lesen auszuwählen. Das hätte den Vorteil, daß bei dynamischer Speicherverwaltung der durch die Pufferverwaltung belegte Speicher immer optimal klein wäre.

Jedoch muß bei allen Methoden, die nicht FIFO organisiert sind bedacht werden, daß Elemente im Speicher veröden können, wenn immer jeweils vorrangige Elemente ausgelöst werden.

Mit anderen Worten: Es kann passieren, daß ein Element nie an die Reihe kommt, weil es immer durch eine noch wichtigere Nachricht verdrängt bleibt. Die FIFO Strategie schließt dies hingegen aus.

Eine weitere Möglichkeit, Elemente vorrangig vor anderen zu behandeln, besteht in der Verwendung von Prioritäten. Beim Schreiben in einen Puffer muß zusätzlich zur Pufferidentifizierung und zum Datenelement noch eine Priorität mit angegeben werden. Bei dieser Steuerung muß jeder Puffer aus mehreren Ebenen bestehen, die jeweils einen Puffer nach FIFO enthalten. *Abbildung 4.5* zeigt einen derartigen Puffer. Dabei wird beim Auslesen immer zunächst bei Priorität 0 nachgesehen, falls dort der Anker gleich NIL ist, geht es eine Stufe abwärts nach 1. Dies erfolgt solange, bis ein gültiger Anker gefunden ist. Dieser zeigt dann auf die Information, die ausgelöst werden soll. Die Vorgehensweise ist ähnlich wie bei der Auswahl von Tasks im Dispatcher.

Eine mögliche Anwendung im Mailbox-Beispiel wäre es, den Puffer der CONSOLE Task so zu organisieren. Die Meldungen von USER Tasks hätten geringe Priorität; hingegen würde man Systemfehlermeldungen, wie „Störung im Telefonmodem" oder ähnliches, mit hoher Priorität an den CONSOLE Prozeß weiterleiten.

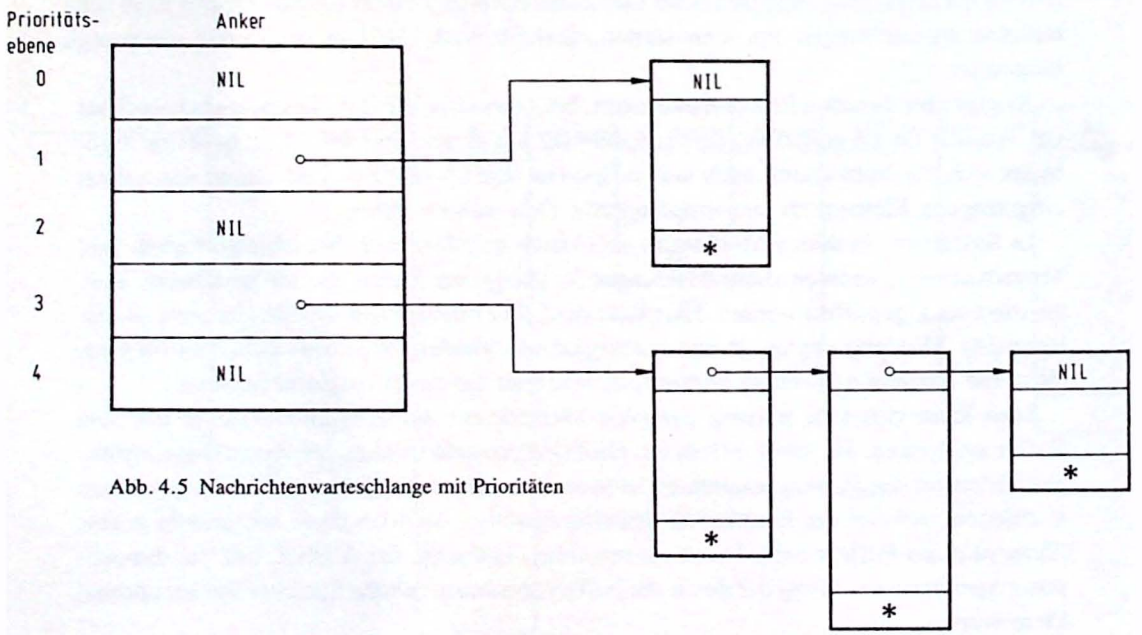


Abb. 4.5 Nachrichtenwarteschlange mit Prioritäten

Die Anzahl der Prioritätsebenen sollte im vernünftigen Umfang liegen; sie sollte also nicht zu groß sein, da das der Übersichtlichkeit des Programms schadet. Jedoch unterstützt die richtige Wahl der Ebenen die Strukturierung des gesamten Problems weitgehend.

Da ein priorisierter Puffer intern mehrere Puffer hat, könnte man auch auf dieses Konzept verzichten und die Priorisierung auf Taskebene erledigen. Dies führt zwar zum selben Ergebnis, es läßt sich jedoch mit Hilfe einer priorisierten Verwaltung eine hierarchische Struktur besser realisieren.

4.6 Prozeßkommunikation mit Antwort

Durch die Aufsplitterung eines Problems in kleinere Teilaufgaben existieren Tasks, die jeweils einen abgeschlossenen Auftrag erledigen. Sie erhalten ihre Order über einen Nachrichtenpuffer, dessen Elemente jeweils einen Auftrag beschreiben. Ist die Fortführung des Auftraggebers von dem Ergebnis der Order abhängig, so muß eine Synchronisation erfolgen. Das bedeutet, daß der Auftraggeber abwarten muß, bis der Auftrag komplett abgearbeitet ist und ein Ergebnis vorliegt (z. B. Sortieren oder Mischen von Datensätzen). Die weitere Diskussion über den Mailbox-Rechner soll dies verdeutlichen.

Jeder registrierte Benutzer einer Mailbox, der sich über eine Benutzernummer sowie ein Paßwort identifizieren muß, verfügt über einen eigenen Speicherbereich, nämlich über seinen Postkasten. In diesen kann jeder andere Benutzer ihm einen Eintrag hinterlassen, der dann zu einem späteren Zeitpunkt herausgenommen wird. Da die Verwaltung der Postkästen eine isolierte Aufgabe darstellt, wird die Ausführung von einem eigenen Prozeß, der hier als BOX bezeichnet wird, übernommen. Er kann von außen einen Schreib- oder Leseauftrag erhalten.

Die USER Prozesse beauftragen den BOX Prozeß über einen Nachrichtenpuffer und erhalten die Antwort (Ergebnis) in einem anderen Puffer zurück. Hierbei ist die Fortführung einer USER Task nur dann sinnvoll, wenn der BOX Prozeß einen Eintrag gelesen und ihn an den Auftraggeber zurückgeschickt hat.

Bei einem Schreibauftrag, der eine Eintragung in die Box veranlaßt, muß der Auftraggeber mitgeteilt bekommen, ob die Information korrekt in der Box abgelegt wurde.

In beiden Fällen wartet der USER Prozeß auf eine Antwort. Da die Laufzeit des BOX Prozesses unbestimmt ist, erfolgt die Synchronisation zweckmäßig durch eine Semaphore *(Abbildung 4.6)*.

Um den Programmierer auf Taskebene von zu vielen Details (Benutzung einzelner Semaphoren) zu entlasten, wird eine Systemprozedur aufgebaut, die das Warten und Fortsetzen nach Erhalt des Ergebnisses regelt. Die Prozedur „send-wait-answer" wird durch Programm 4.7 allgemein definiert.

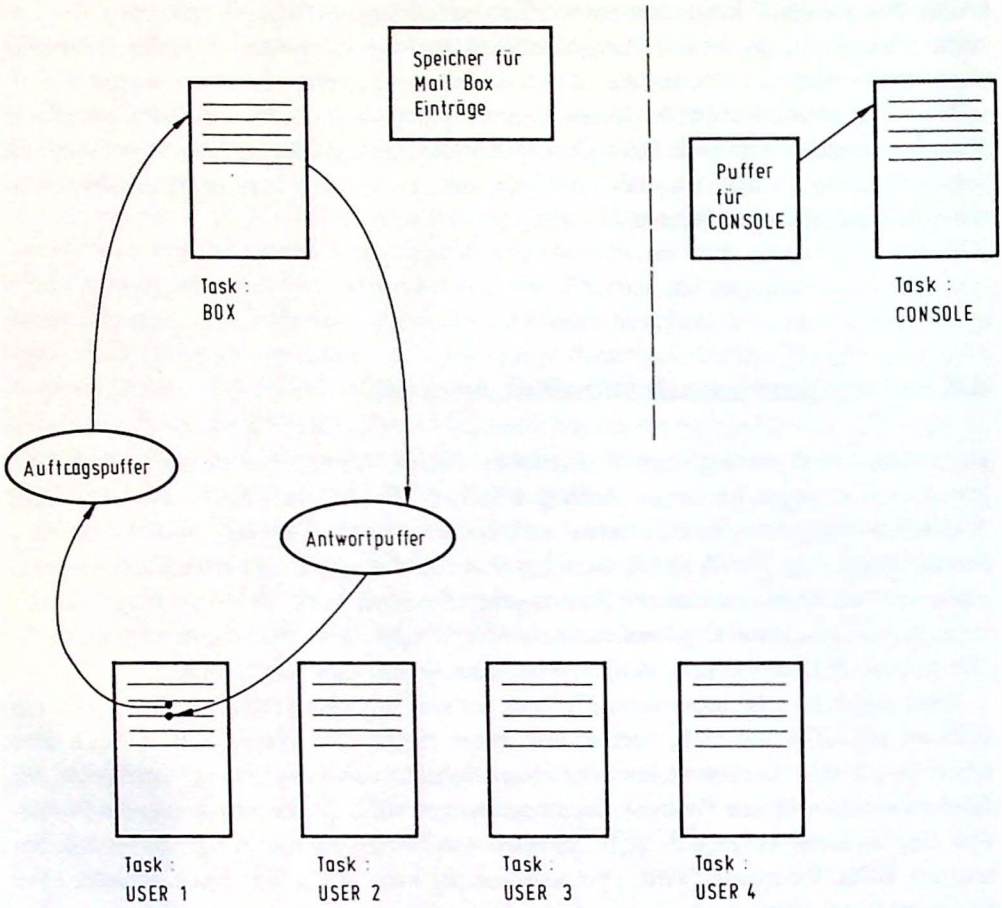


Abb. 4.6 Pufferung mit Warten auf Antwort

Prozedur send_wait_answer(job,answer)

. . .

Sende job in den Auftragspuffer.

wait (antwort) /Semaphore antwort regelt Fortführung von sender/

Lese aus Antwortpuffer und speichere Ergebnis nach answer.

. . .

Prozedur answer /wird von BOX aufgerufen wenn Ergebnis vorliegt/

. . .

```
Sende Ergebnis in Antwortpuffer.
signal (antwort)
...
init(antwort,Ø)
```

Programm 4.7

Durch den Aufruf von „send_wait_answer" wird ein Auftrag abgesendet und der Auftraggeber verzögert. Der Prozeß BOX sendet durch den Aufruf von „answer" die gefundene Information oder eine Fehlermeldung an den USER Prozeß zurück, der durch ein „signal" auf antwort fortgeführt wird.

Die Datenstruktur solcher Puffer besteht also aus zwei herkömmlichen Nachrichtenpuffern, die intern zwei Semaphoren (siehe Kapitel 4.1) besitzen. Außerdem verfügen sie über eine weitere Semaphorvariable zur Synchronisation auf die Antwort.

Die Organisation der Puffer (ob Ring oder Queue) ist unerheblich, jedoch müssen sie nach FIFO arbeiten. Die beschriebene Kommunikationsform unter Tasks wird verbreitet auch als Rendezvous bezeichnet, da eine Task auf die andere wartet, bis sie sich treffen (synchronisieren).

Es ist bemerkenswert, daß sich auch diese komplizierteren Puffermethoden alle auf Semaphoren zurückführen lassen.

4.7 Ringpuffer unter BSØ

Da die Anforderung an BSØ einerseits darin bestand, ein sehr kompaktes und schnelles System zu schaffen und andererseits doch gewisse Grunddienste, so auch eine Pufferverwaltung zu integrieren, wurden Ringpuffer gemäß Kap. 4.3 eingesetzt. Sie arbeiten sehr sicher und sind einfach zu implementieren.

Es sind zwei Routinen für die Taskebene zugänglich. Sie dienen zum Senden und Empfangen von Nachrichten („send" und „receiv").
Bei der Assemblierung des Betriebssystems, bei der es zugleich konfiguriert wird, ist die Anzahl der gewünschten Puffer unter NPUF, die Anzahl der Bytes pro Element unter PUFLNG und die maximale Anzahl von Elementen (Größe des Rings) unter MAXENT anzugeben. Gemäß dieser Parameter werden die Puffer im Datenbereich ab Label PUFERS angelegt.

Wird für NPUF eine Drei angegeben, so stehen die Puffer Ø,1 und 2 zur Verfügung. Der erste Puffer hat also die Nummer Ø, der letzte die Nummer NPUF-1.

Die Anordnung der Information wird durch *Abb. 4.7* illustriert.

Da jeder Puffer zwei eigene Semaphorvariablen benötigt, werden sie mit ihren Nummern einfach an die Benutzer-Semaphoren angehängt. Ihre Nummern beginnen demnach bei NSEMAS. Die zusätzlichen Semaphoren werden automatisch durch die Konfiguration eingerichtet und haben folgende Zuordnung zu den Puffern:

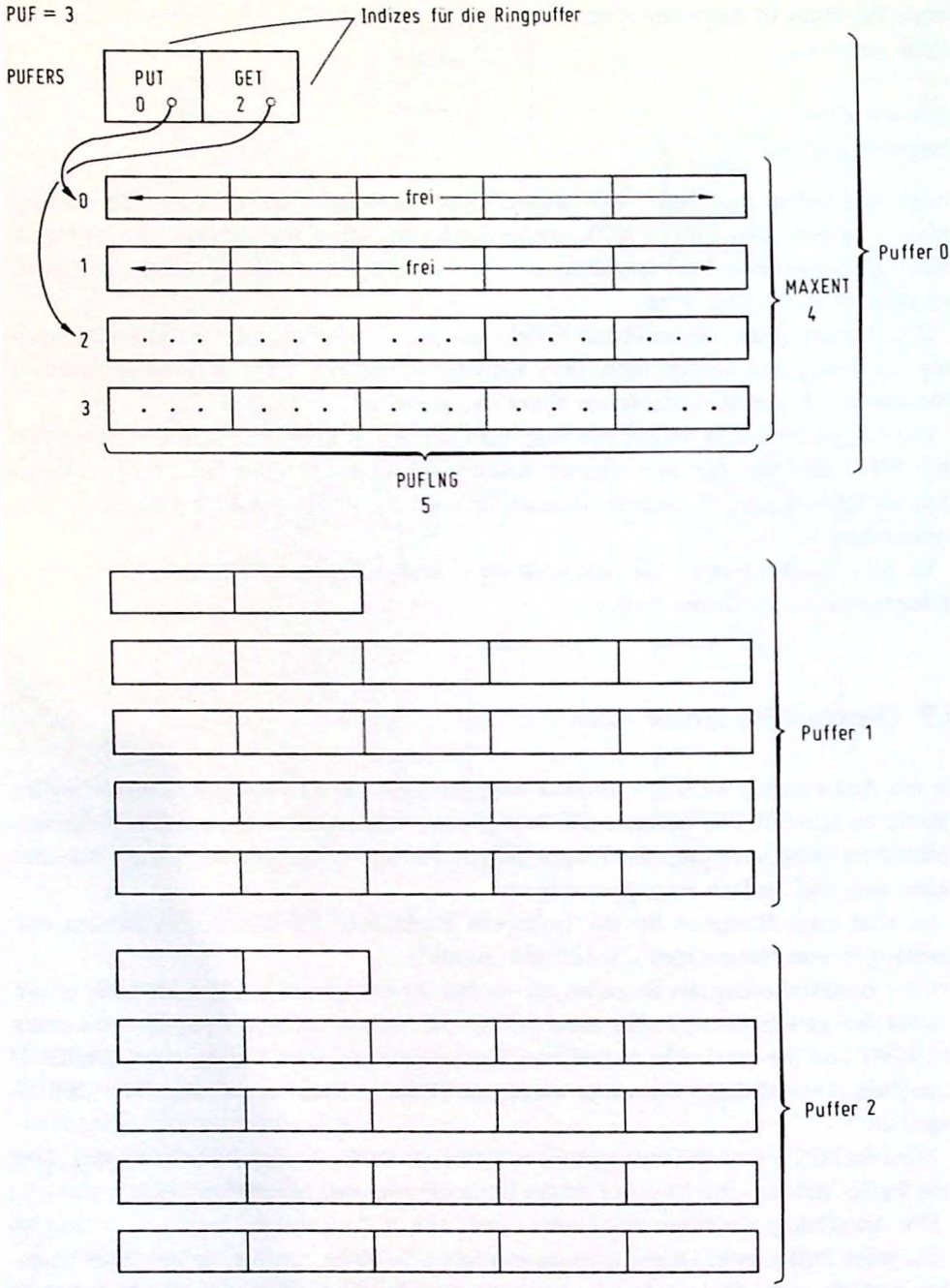


Abb. 4.7 Datenstruktur der BSØ Ringpuffer

	Semaphorennummer
Puffer 0	NSEMA und NSEMA+1
Puffer 1	NSEMA+2 und NSEMA+3
...	...

Bei der Konfiguration des Systems ist darauf zu achten, daß maximal 255 Semaphoren im System möglich sind. Man muß also pro Puffer zwei Semaphoren mit einberechnen! Da die Semaphoren beim Urstart nicht initialisiert werden, ist eine Prozedur nötig, die das für die Semaphoren der Pufferverwaltung ausführt. Die Prozedur heißt INIPUF und wird zu Beginn aufgerufen.

Beim Senden und Empfangen, was durch den Aufruf von „send" bzw. „receiv" eingeleitet wird, ist im Akku die Nummer des Puffers und im BC Registerpaar ein Pointer auf die Information bzw. auf den Datenbereich, wo die Information hingeschrieben werden soll (im Falle von „receiv") anzugeben.

Zusammenfassung

Die Prozeduren zum Austausch von Daten unter einzelnen Tasks sind ein wesentlicher Bestandteil eines Multitasksystems. Zur Pufferung von mehreren Elementen pro Puffer, was normalerweise der Fall ist, gibt es zwei grundlegende Methoden.

Die Ringpufferverwaltung hat einen fixen Datenbereich, da sie die Elemente „ringmäßig" ablegt. Es ist eine relativ einfache Datenstruktur, die kompakt und schnell zu implementieren, vor allem sehr angenehm auszutesten ist.

Die Nachrichten-Queues arbeiten nach einem dynamischen Prinzip, bei dem die Datenbereiche zwischen Pufferverwaltung und Task nicht unbedingt abgegrenzt sind. Die Methode ist sehr speichereffizient, braucht aber, um korrekt zu arbeiten, eine dynamische Speicherverwaltung, von der man sich je nach Bedarf Speicherbereiche zuteilen lassen kann. Dies hat einen höheren Software-Aufwand zur Folge.

Bei beiden Methoden gibt es den Fall, daß eine Task in einen Puffer schreiben will, dieser jedoch voll ist. Die Task muß solange verzögert werden, bis wieder ein Element frei wird. Dieses Freiräumen erfolgt durch einen „receive" Aufruf, der seinerseits die wartende Task fortsetzt. Zu diesem Zweck bietet sich die Verwendung von Semaphoren an, die das Problem elegant lösen.

Ein anderer Fall, der ebenfalls zur Verzögerung einer Task führt, ist das Lesen eines leeren Puffers. Auch hier wird eine Task, die Daten verlangt, aber noch keine vorhanden sind, bis zum Eintreffen einer Sendung verzögert. Dazu dient eine weitere Semaphore.

Daneben existieren noch einige Versionen von Puffern, die nach anderen Prinzipien (LIFO, Prioritäten etc.) verwaltet werden, sowie ein komplizierteres Verfahren der Pufferung, das eine Antwort einer Task mit in den Ablauf einschließt.

5 Interruptverwaltung

Sollen Ereignisse, die zu nicht vorhersehbaren Zeitpunkten eintreten, mit einem Mikroprozessor kontrolliert werden, so kann derer Eintreffen dadurch festgestellt werden, daß zyklisch nachgesehen wird, ob sie eingetreten sind oder nicht.

Da von einer Abfrage zur anderen eine bestimmte Zeit, die dadurch bestimmt ist, was der Prozessor sonst noch zu tun hat und wie lange er dazu braucht, vergeht, kann es einerseits dazu kommen, daß sehr kurze Ereignisse nicht registriert werden, oder daß auf Ereignisse zu langsam reagiert wird.

Diese Methode, die man auch als Polling bezeichnet, hat zudem den offensichtlichen Nachteil, daß durch das zyklische Abfragen von Signalen ständig Befehle abgearbeitet werden, die in nur sehr wenigen Fällen zu einem positiven Ergebnis führen. Sie lösen nur dann eine sinnvolle Aktion aus, wenn das Signal eingetreten war. Diese Technik wird z. B. für die Abfrage von Tasten verwendet; sie werden dann oft stundenlang nicht gedrückt, aber tausende Male vom Prozessor abgefragt!

Dies mutet widersinnig an und rechtfertigt sich nur damit, daß der Rechner, da er von Haus aus schnell ist, dies ohne weiteres zusätzlich ausführen kann, und daß diese Methode zuverlässig arbeitet.

Kritischer jedoch ist der Fall, wenn eine bestimmte Reaktionszeit auf das Ereignis gefordert wird. Hier muß man so programmieren, daß diese eingehalten wird. Dabei kann es zu einem unübersichtlichen Programm kommen, da man in der Hauptschleife gegebenenfalls öfters das Signal abfragen muß. Besonders gefährlich wird es, wenn mehrere zeitkritische Ereignisse gehandhabt werden müssen. Eine effizientere Vorgehensweise bei diesen Problemen erreicht man durch die Verwendung von Interrupts, die bereits oben (Kapitel 2) für eine korrekte Zeitsteuerung verwendet wurden.

Hardware-mäßig existieren für diese Zwecke ein oder mehrere Eingänge am Prozessor. Wird ein solcher Eingang mit einer Signalleitung belegt, so unterbricht der Prozessor beim Auftreten des Signals sein gerade laufendes Programm und verzweigt auf eine definierte Adresse, bei der das Interrupt Service Programm beginnt. Durch den Interrupt wird die Rückkehradresse (die Stelle, wo das Programm unterbrochen wurde) automatisch gesichert. Dadurch findet der Prozessor, nach Abhandlung der Interrupt Service Routine, wieder die Programmstelle, wo er unterbrochen wurde. Dies ist ein prinzipieller Mechanismus und ist nicht direkt abhängig von einem Multitasksystem.

Fragt man nun z. B. eine Taste nicht zyklisch ab, sondern läßt sie einen Interrupt auslösen, so wird der Rechner nur dann beansprucht, wenn auch wirklich ein Ereignis eintrat. Der Vorteil ist offensichtlich.

5.1 Interrupts schnell aber gefährlich

Da Interrupts direkt vom Auftreten sporadischer Ereignisse abhängen, ist ihr Gebrauch oft nicht trivial. Ist nur ein einziger Interruptgrund vorhanden, z. B. Uhrinterrupt, so ist dieser relativ einfach zu überschauen. Hingegen ist, bei einer Vielzahl von verschiedenen Interrupts, das Testen des Systems sehr schwierig, besonders dann, wenn ein Interrupt von einem anderen abhängig sein kann. Das ist z. B. der Fall bei diversen Bausteinen, die eine serielle Schnittstelle interrupt-gesteuert betreiben.

Sie lösen einen Interrupt aus, wenn ein Zeichen empfangen wurde, aber auch, wenn ein Partity Fehler vorlag. Braucht man zudem noch weitere externe Geräte (z. B. für präzise Steuerungen) und lösen diese ebenfalls Interrupts aus, so wird die Testbarkeit des Programmes dadurch erschwert, daß während einer Interrupt Service Routine ein anderer Interrupt zuschlagen kann, und in diesem wieder ein anderer. Dabei kommt es in der Praxis oft zu unerwünschten Ergebnissen, obwohl auf dem Papier alles durchweg plausibel wirkt. Man muß dann teurere Testmethoden anwenden (Emulatoren, Logikanalysator usw.), was sich als zeitaufwendig und nicht immer einfach gestaltet.

Durch diese Nachteile erkauft man sich aber auf jeden Fall ein System mit kurzen Reaktionszeiten. Deshalb setzt man die Interruptverarbeitung vor allem bei Problemen ein, die Realzeitbedingungen unterliegen.

Man sollte also bei der Planung abwägen, welche Aktionen wirklich zeitkritisch sind, und welche auch ohne Interrupt, eventuell mit größerem Software-Aufwand, abgehandelt werden können.

5.2 Methoden der Interruptverwaltung

Alle handelsüblichen Mikroprozessoren verfügen über mindestens einen Interrupteingang, über den die CPU unterbrochen werden kann. Einfache Prozessortypen rufen beim Auftreten des Interrupts eine Prozedur auf, deren Adresse festliegt. Es ist also nur eine Routine vom Interrupt direkt auslösbar.

Sollen bei diesen Prozessen verschiedene Geräte Interrupts auslösen können, so muß die Interrupt Service Routine zunächst abfragen, welche externe Einheit für den Interrupt verantwortlich ist und muß dann anhand ihrer Ermittlungen weiter entscheiden, welcher Programmteil ausgeführt wird.

Das Abfragen nach dem Interruptverursacher entfällt bei sogenannten vektorisierten Interrupts. Dazu benötigt man jedoch spezielle externe Bausteine, die zusätzlich zum Interrupt Signal der CPU einen Vektor auf dem Datenbus übergeben. Die CPU ermittelt dann automatisch eine Adresse, bei der sie die entsprechende Interrupt Service Routine erwartet.

Die Mikroprozessoren der 16-Bit-Gruppe und aufwärts, verfügen in der Interruptverwaltung über einige weitere Möglichkeiten. So existieren bei etlichen Typen zusätzlich zu den beschriebenen Interrupts, die von externen Geräten stammen, noch interne, die dann ausgelöst werden, wenn Laufzeitfehler auftreten (z. B. Division durch Null). Sie werden auch als Trap bezeichnet.

Neben diesen Interrupts gibt es häufig noch einen zweiten Typ, der als NMI (Non maskable Interrupt) bezeichnet wird. Er hat prinzipiell die gleiche Wirkung, mit dem Unterschied, daß er im Falle einer Unterbrechung auf eine andere Adresse springt. Außerdem ist er gegenüber dem normalen Interrupt software-mäßig nicht zu sperren; d. h. er ist ständig scharf. Der NMI wäre also für den Uhrinterrupt nicht brauchbar. Hingegen setzt man ihn gern als Notbremse ein; z. B. dann, wenn die Versorgungsspannung unter den vertretbaren Wert sinkt und noch irgendwelche wichtigen Aktionen (retten von Daten in ein batteriegepuffertes RAM) zu tun sind.

5.3 Zentrale Interrupt-Verwaltung im Systemkern

Das bisher über Interrupts Gesagte gilt allgemein und steht nicht im direkten Zusammenhang mit einem Multitaskkern. Sollen Interrupts in einem Multitaskkern benutzt werden, so ist es die Aufgabe eines zentralen Interrupt-Verwaltungsprogramms, dem Interrupthandler, Interruptquellen zu ermitteln und entsprechende Aktionen einzuleiten.

Da die Anzahl der Interrupts und die Struktur schlechthin sehr von der Aufgabenstellung des Gesamtsystems abhängt, ist es schwer, einen konsistenten Weg zu finden. Jedoch kann man festhalten, daß die Anzahl der möglichen Interrupts, sowie die zu dem einzelnen Interrupt korrespondierenden Routinen, bei der Konfiguration bzw. beim Initialisieren dem Systemkern zu übergeben sind. D. h. man muß dem System einmalig sagen, welche Service Routine nach Auftreten eines bestimmten Interrupts abzuhandeln ist. Diese Information ist unbedingt für den Interrupthandler nötig.

Geht man von der einfachsten Interruptstruktur, die bei jedem Mikroprozessor möglich ist, aus, so hat der Interrupthandler eines Multitaskkerns folgende Schritte durchzuführen.

1) Sichern der Registerstände einer Task

Im Gegensatz zu einem sequentiellen Programm, muß unter Mehrprogrammbetrieb der komplette Registersatz einschließlich des Stack Pointers gesichert und die aktuelle Task von „running“ nach „ready“ überführt werden. Es ist nämlich nicht gewiß, ob diese Task nach der Interrupt-Abhandlung auch wirklich als „running“ Task weiterlaufen darf. Die Task wird also nicht einfach unterbrochen, sondern wird in die ready Warteschlange des Prozessors eingereiht. An ihr wird erst dann weitergearbeitet, wenn der Dispatcher ihr den Prozessor zuweist.

2) Grund für die Unterbrechung ermitteln

Nach dem Aufzeichnen der Registerzustände ermittelt der Interrupthandler den Grund des Interrupts. Dies kann z. B. durch einen Lesebefehl auf einen bestimmten Eingang geschehen, wobei der gelesene Wert Rückschlüsse auf das Gerät, das die Unterbrechung ausgelöst hat, zuläßt.

Handelt es sich um den Uhrinterrupt, der für die gesamte Zeitsteuerung nötig ist, muß der Handler in den Scheduler verzweigen. Ist es hingegen ein anderer Interrupt, wird die Service Routine aufgerufen, die bei der Initialisierung diesem Ereignis zugewiesen wurde. Sie bedient im weiteren Verlauf das anfordernde Gerät.

Solche Prozeduren werden allgemein als Treiber bezeichnet, besitzen aber dann noch zusätzliche Merkmale, die später aufgegriffen werden.

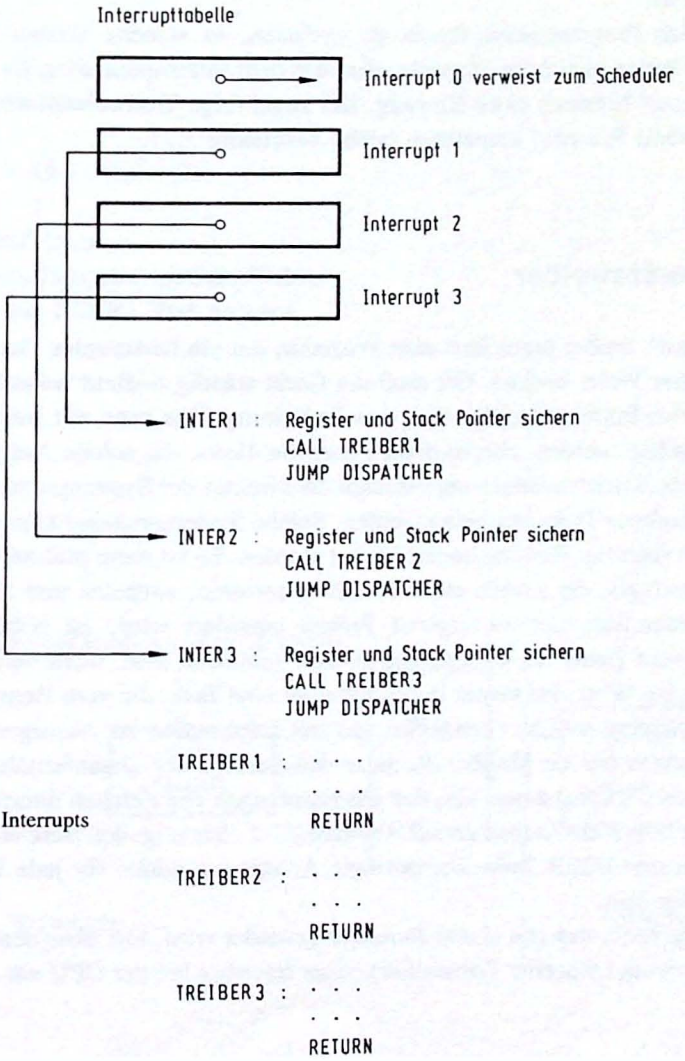


Abb. 5.1 Vektorisierte Interrupts

3) Rückkehr in den normalen Ablauf

Nachdem aus einem Treiber (oder auch dem Scheduler) zurückgekehrt wurde, wird, um die Interruptbehandlung abzuschließen, der Dispatcher aufgerufen, der seinerseits die Task startet, die als nächste an der Reihe ist.

Stehen vektorisierte Interrupts zur Verfügung, so ist eine der Hauptaufgaben des Handlers schon auf den Prozessor übertragen. Ein Interrupt schlägt dann nicht zentral ein, sondern direkt beim entsprechenden Treiber.

Bei der Initialisierung müssen die Treiberadressen an die richtige Stelle der Vektorpage geschrieben werden. Der Treiber selber muß zu Beginn die Register der Task in ihrem privaten Stack sichern und kann sich dann erst seinen Aufgaben widmen. Ist der Treiber fertig, so begibt er sich, wie oben, in den Dispatcher, um eine Task auszuwählen und fortzusetzen.

Um den Programmierer davon zu entlasten, in seinem Treiber Registerinhalte in richtiger Weise zu sichern, überträgt man das dem Interrupthandler. Der Handler hat dann pro Interrupt-Nummer einen Eingang. Der zugehörige Treiber wird vom Handler aus wie eine normale Prozedur aufgerufen (siehe *Abbildung 5.1*).

5.4 Gerätetreiber

Der Begriff Treiber bezeichnet eine Prozedur, die ein bestimmtes Gerät betreibt, also in irgendeiner Weise bedient. Oft muß ein Gerät ständig bedient werden, z. B. zyklisches Senden von Information über eine serielle Leitung. Das kann mit Interrupts alleine nicht bewerkstelligt werden. Hierzu dienen normale Tasks, die solche Aufgaben durchführen.

Größere Betriebssysteme unterstützen die Struktur der Systemprozesse, welche vorrangig vor anderen Tasks bearbeitet werden. Solche Systemprozesse können auch dynamisch aus einer Interrupt Routine heraus kreiert werden. Es ist dann praktisch, wenn kurzzeitig viele Interrupts, die jeweils einen Prozeß generieren, auftreten und Daten erzeugen. Da durch jeden Interrupt ein eigener Prozeß generiert wird, ist sichergestellt, daß die angefallenen Daten, da sie an einen Prozeß gebunden sind, nicht verlorengehen.

Einfacher ist es, aus einem Interrupttreiber eine Task, die vom Benutzer (Programmierer) eingerichtet wurde, anzustoßen und mit Information zu versorgen.

Betrachten wir im Mailbox-Rechner den Bereich der Datenfernübertragung genauer, und gehen wir dabei davon aus, daß das Empfangen von Zeichen interrupt-mäßig erfolgen soll, ergibt sich ein Aufbau gemäß *Abbildung 5.2*. Sie zeigt den Bereich der Kopplung von Benutzer und USER Task. Die gezeigte Anordnung müßte für jede USER Task einmal vorhanden sein.

Ein Zeichen, das von einem Benutzer gesendet wird, löst über den UART (Universal Asychnchronus Receive Transmitter) einen Interrupt bei der CPU aus. Angenommen, es

sind keine vektorisierten Interrupts vorhanden, so verzweigt die CPU in den Interrupthandler des Multitasksystems. Dort wird festgestellt, nachdem die Register gesichert wurden, daß der Interrupt vom UART1 kam, also wird auf den entsprechenden Treiber verzweigt. Der Treiber hat die Aufgabe, das empfangene Zeichen aus dem UART auszulesen und an die USER1 Task irgendwie weiterzuleiten. Zuvor soll der Treiber das Zeichen dem Mailbox-Benutzer zurückschicken, damit er eine positive Rückmeldung erhält. Die Übergabe der Information an die USER1 Task kann durch die Synchronisation über eine Semaphore erfolgen.

```
Task USER1
...
init(0)
...
wait
...
Zeichen aus dem Puffer lesen
...
Interrupt Service UART /Treiber1/
...
Zeichen vom UART lesen
Zeichen an Mailbox-Benutzer zurückschicken
Zeichen in Puffer für USER1 Task ablegen
...
signal_from_interrupt
...
```

Programm 5.1

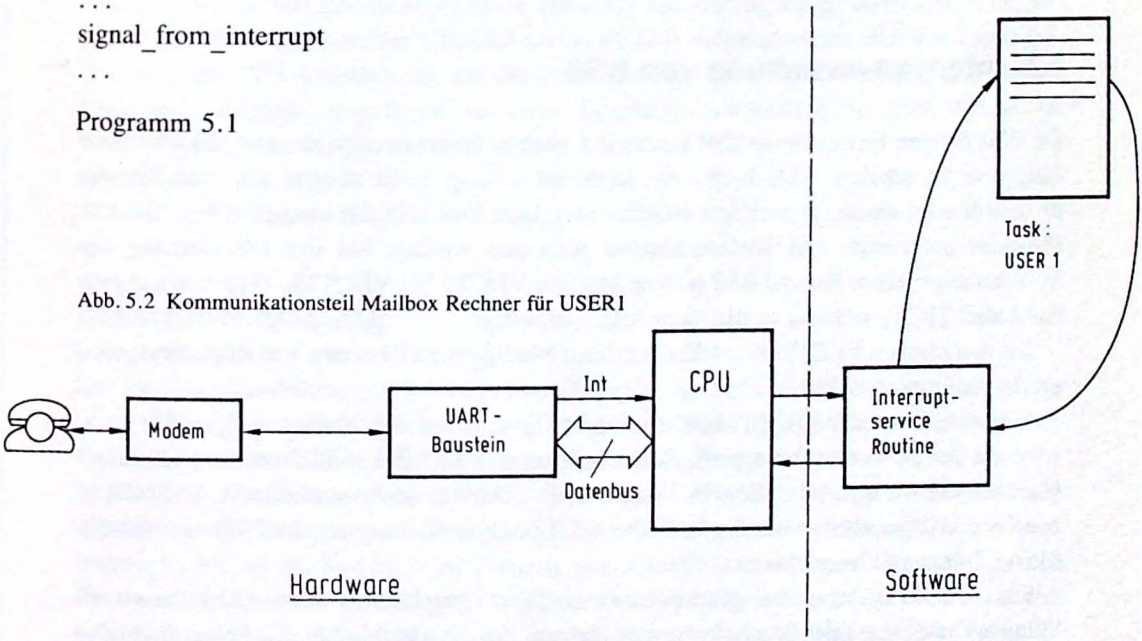


Abb. 5.2 Kommunikationsteil Mailbox Rechner für USER1

Programm 5.1 zeigt eine mögliche Lösung. Sollen „signal" Aufrufe von der Interrupt-Ebene aus erfolgen, so ist dazu eine spezielle Prozedur „signal_from_interrupt" einzusetzen. Das ist nötig, weil der normale „signal" Aufruf, nachdem die Operation ausgeführt wurde, nicht mit einem normalen Return zurückkehrt, sondern über den Dispatcher den weiteren Ablauf einleitet.

Das ist hier nicht wünschenswert, da aus der Semaphorenprozedur „signal_from_Interrupt" in die Interrupt Routine zurückgekehrt werden muß. Denn es kann sein, daß der Treiber noch weitere Befehle abzuhandeln hat.

Die Prozedur „signal_from_interrupt" hat also nur den Zweck, eine gegebenenfalls wartende USER Task in den Zustand „ready" zu versetzen, um sie damit im weiteren Verlauf zu starten.

Überhaupt ist es sicherer, die Synchronisation von Prozeß und Interrupt durch einen Nachrichtenpuffer, z. B. Ringpuffer, vorzunehmen. Auch hier darf nicht die normale „send" Prozedur verwendet werden, da im Falle, daß der Puffer voll ist, die Service Routine verzögert werden müßte. Dies aber geht nicht mit Interrupt Routinen, da sie keinen eigenen Stack und Deskriptorblock haben. Es muß demnach eine Prozedur geschrieben werden, die die Charakteristik, der „send_no_wait" besitzt und darüber hinaus, intern die Prozedur „signal_from_interrupt" zum Anstoßen von wartenden Prozessen verwendet. Problematisch ist nur der Fall, wenn die Service Routine auf einen vollen Puffer tritt. Hier muß eine geeignete Fehlerbehandlung eingreifen, da eine Verzögerung auf Interruptebene nicht möglich ist.

5.5 Interruptverwaltung von BS0

Da BS0 für den Prozessortyp Z80 konzipiert wurde, bietet es sich an, mit vektorisierten Interrupts zu arbeiten. Das heißt, ein Interrupt schlägt nicht zentral ein, sondern der Prozessor wird direkt, je nachdem welcher periphere Baustein ihn ausgelöst hat, auf eine Prozedur aufgesetzt. Die entsprechenden Adressen werden bei der Übersetzung des Systems angegeben. Es sind BS0 interne Marken VECT2 bis VECT10. Daneben existiert das Label TICK, welches in den Scheduler verzweigt.

Bei den Marken VECT2 etc. stehen zunächst Macros zur Sicherung von Registerständen sowie des Prozeßstacks.

Bevor die Prozedur TREIB2 (die Zwei gibt Hinweis auf den Vektor) aufgerufen wird, wird auf den Systemstack umgeschaltet. Die Prozedur TREIB2 muß vom Programmierer bereitgestellt werden und bedient im Endeffekt das interrupt-auslösende Gerät. Dabei ist zu beachten, daß im Treiber nur die Prozedur SIGFI aufgerufen werden darf, da sie exklusiv für die Interrupt-Ebene existiert.

Ein vom Programmierer geschriebener Treiber braucht sich also nicht darum zu kümmern, Register oder Stackadressen zu sichern, sondern beinhaltet die Abhandlung des

Interrupt-Auslösers. Er kann nur durch SIGFI mit Tasks Kontakt aufnehmen; eine andere Möglichkeit ist nicht vorgesehen.

Bei der Assemblierung des Betriebssystems, die zugleich die Konfiguration darstellt, sind gemäß des Problems, die Marken VECT4 bis VECT10 zu ergänzen. Dies richtet sich jedoch sehr stark danach, welche Interrupts im Gesamtsystem bedient werden sollen. Soll z. B. der Vektor 8 zusätzlich auftreten, so ist die Adresse VECT8 analog VECT2 zu besetzen.

```
VECT8:    DI
          RSAVE
          SPSAVE
          LD SP,(SYSSTK)
          CALL TREIB8
          SPRELO
          RRELOD
          RETI
```

Programm 5.2

Vom Benutzer muß selbstverständlich dann eine Treiberroutine TREIB8 bereitgestellt werden, die den Interrupt mit dem Vektor 8 abhandelt.

Da alle Bausteine, die den Interruptmodus 2 des Z80 unterstützen, initialisiert werden müssen, ist eine entsprechende Aktion einmalig am Anfang nötig. Hierzu wird bei der Marke POWUP die Prozedur CLKINI sowie PIOINI aufgerufen, die hier den Timerbaustein und eine PIO initialisieren. Da die Initialisierung immer direkt von den verwendeten Bausteinen abhängt, wurde sie aus dem Systembereich herausgenommen und in die Verantwortung des Programmierers gelegt.

Zusammenfassung

Die Interruptverarbeitung ist eine probate Methode, um auf sporadisch auftretende Ereignisse mit extrem kleiner Antwortzeit zu reagieren. Dazu ist jedoch eine entsprechende Hardware-Logik, die heute in jedem handelsüblichen Prozessor vorhanden ist, nötig.

Mit dem Einsatz von Interrupts handelt man sich nicht nur Vorteile ein, denn die Testbarkeit des Gesamtsystems wird erschwert. Dies wird besonders kritisch, wenn Interrupts von anderen (d. h. vom vorherigen aufgetretenen) abhängen. Deshalb ist beim Systementwurf genau abzuwägen, welche Probleme ohne Interrupt nicht oder nur mit hohem sonstigen Aufwand, zu lösen sind; nur diese sind für Interrupts vorzusehen.

5 Interruptverwaltung

In einem Multitasksystem liegt die oberste Gewalt über Interrupts in einem Interrupthandler, der je nach System und Prozessor komplex ausfällt. Er sorgt dafür, daß beim Eintreten eines Interrupts die Registerstände der unterbrochenen Task sauber gesichert werden und verzweigt anschließend in die einzelnen Interruptprogramme, die oft als Treiber (wenn sie externe Geräte bedienen) bezeichnet werden. Die Selektion der Routine, bei der es sich auch um den Scheduler / Dispatcher im Falle des Zeitinterrupts handeln kann, erfolgt entweder software-mäßig durch Abfragen des Auslösers oder hardwaremäßig über vektorisierte Interrupt-Logik. Die Synchronisierung einer Task auf einen Interrupt kann durch eine Semaphore erfolgen, wobei spezielle Prozeduren von der Interrupt-Ebene verwendet werden müssen.

6 Fehlerbehandlung

Da in der Regel jedes fertige Programm in irgendeiner Weise mit Fehlern behaftet ist, und da während des Betriebes eines Systems Störungen auftreten können, ist es wichtig, Fehler oder Störungen zu erkennen und auf diese durch Gegenmaßnahmen zu reagieren.

Ein fertiges Programm ist meistens deshalb noch mit Fehlern behaftet, weil es nicht effektiv wäre, eine Entwicklung jahrelang im Labor zu testen, bis man sicher sein kann, daß kein Fehler mehr vorhanden ist. Vielmehr verlegt man aus technischen und kommerziellen Gründen den letzten Teil des Tests in die Erprobungsphase und erhält dadurch vom Endverbraucher Angaben über Fehlfunktionen des Systems (Nullserientest).

Um so arbeiten zu können, ist es unabdingbar, ein System zu schaffen, das Fehler verkraftet und diese gegebenenfalls korrigiert. Es wird dann als fehlertolerantes System bezeichnet. Diese Systeme zeichnen sich dadurch aus, daß der Programmlauf nicht durch Auftreten eines Fehlers zusammenbricht, der eigentlich gar nicht so fatal war.

Zentrales Problem ist hier die Erkennung von Fehlern. Bevor wir dieses Thema näher betrachten, kategorisieren wir die Fehlergründe, die hier von Bedeutung sind.

6.1 Fehlerarten

Eine sinnvolle Gliederung erfolgt in:

- Logische Fehler
- Programmierfehler
- Benutzerfehler
- Hardware-Fehler

- *Logische Fehler*

Logische Fehler können nicht, oder nur mit sehr hohem Aufwand, erkannt werden. Die Erkennung und Beseitigung dieser muß vom Programmierer aus erfolgen. Es handelt sich hier um Fehler, die in der Programmlogik stecken und daher rühren, daß Problemlösungen nicht korrekt in ein Programm umgesetzt wurden.

- *Programmierfehler*

Bei diesem Typ wurden Fehler bei der Kodierung des Programms gemacht. Es kann sich hier um Syntax-Fehler handeln oder um einen formal falschen Aufruf von Prozeduren. Die Fehler sind meistens leicht zu erkennen, besonders bei Syntax-Fehlern ist dies einfach. Systemprozeduren sollen, da sie falsch aufgerufen werden können, immer eine Rückmeldung „ok" oder „Fehler" liefern. Es liegt dann beim Programmierer, inwieweit er auf die Rückmeldung eingeht.

- *Fehler durch Benutzer*

Die Fehler dieser Klasse entstehen auf oberster Ebene und werden vom Endbenutzer verursacht. Sie sind durch geeignete Programmierung, z. B. indem man dem Benutzer vorschlägt, was er tun kann *(Help Menü)*, abzufangen.

- *Hardware-Fehler*

Sie werden durch irgendeinen Defekt in der Elektronik verursacht. Auch bei Ausfall von irgendwelchen Bauteilen muß gewährleistet sein, daß das System nicht komplett zusammenbricht, sondern in seinem Leistungsumfang nur eingeschränkt wird.

Ein Multitasksystem vereinfacht die gesamte Problematik grundlegend. Tritt ein Fehler in einer Task auf, so hat er nur lokal zur Task ernste Folgen. Die anderen Tasks können, sofern sie nicht mit der fehlerhaften Task kooperieren, weiterlaufen, was der Stabilität des Systems zugute kommt.

6.2 Fehlerbehandlung

Die Fehlerbehandlung gliedert sich in drei grundsätzliche Schritte.

1) Fehlererkennung

Die Erkennung von Programmierfehlern erfolgt bei Syntax-Fehlern durch den Übersetzer. Bei anderen Programmierfehlern erfolgt ihre Erkennung durch Testläufe. Wie oben erwähnt, sollen Prozeduren vor fehlerhaftem Aufruf geschützt werden. So wäre z. B. ein Überlauf einer Semaphorvariablen in geeigneter Form dem aufrufenden Programm zurückzumelden. Die Rückmeldung erfolgt über einen Fehlerkode, der Rückschlüsse auf den Fehlergrund zuläßt.

Die Fehler durch falsche Benutzereingaben sind von der entsprechenden Task selber zu erkennen. Diese Plausibilitätskontrolle kann sich z. B. auf einen Wert beziehen, der nur in einem gewissen Bereich zulässig ist. Außerhalb dieses Wertes ist eine Eingabe abzulehnen und dieselbe nochmal zu provozieren. Der exemplarische Mailbox-Rechner soll über eine gewisse Bediensprache verfügen. Sie hat eine einfache Syntax, ähnlich einer Kommandosprache, die genau darlegt, was als mögliche Eingabe zulässig ist. Anhand ihrer Definition wird ein eingegebenes Kommando auf seine formale Richtigkeit geprüft, und es wird nur ausgeführt, falls die Eingabe richtig war.

Weitaus komplizierter ist die Überprüfung der systemeigenen Hardware. Dabei ist von einer hardware-mäßigen Lösung abgesehen, ein Power-Up-Test eine übliche Methode. Nach dem Einschalten der Hardware laufen Testprogramme ab, die z. B. den Speicher auf seine Funktion überprüfen. Das kann dadurch erfolgen, daß ein Bitmuster zunächst in den Speicher geschrieben wird. Danach wird es wieder zurückgelesen und mit dem geschriebenen Muster verglichen. Weitere Systemkomponenten, wie Ein-/Ausgabe Bausteine lassen sich auf ähnliche Weise überprüfen.

Dieser einmalige Hardwaretest ist jedoch bei vielen Systemen nicht ausreichend, vielmehr soll ein zyklischer Test erfolgen. Dazu bietet sich eine Task an, die mit geringer Priorität läuft und so den übrigen Rechenbetrieb nicht durch Kontrollen hemmt.

Gerade bei Systemen, die mit anderen kommunizieren, ist eine Überprüfung der empfangenen Information wichtig. Dazu dienen probate Methoden der Fehlererkennung wie Paritätsprüfung sowie CRC-Verfahren.

2) Auswahl der auszuführenden Aktion

Nachdem ein Fehler erkannt ist, muß eine bestimmte Fehlerbehandlung ausgewählt werden. Dazu ist eine Einteilung in Fehlerkategorien nützlich. *Abbildung 6.1* zeigt die Kategorien für den Mailbox-Rechner. Jede Kategorie symbolisiert das Gewicht eines Fehlers. Im Multitasksystem ist es günstig, für die Auswahl der Aktion eine Task zu generieren, die anhand der Fehlerkategorien die Aktionen einleitet.

3) Reaktion auf Fehler

Die Reaktion auf Fehler kann sehr vielschichtig sein. Kategorie 1 würde im Beispiel dazu führen, daß ein Text oder ein gewisses Signal dem Benutzer mitteilt, daß ein Hardware-Fehler vorliegt, ohne dessen Behebung ein Fortfahren nicht sinnvoll ist. Der Prozessor ist in einem definierten Zustand zu stoppen (HALT), da nichts sinnvolles mehr ausgeführt werden kann. Davor sollte auf dem Bildschirm ein Protokoll abgesetzt werden, das detaillierte Rückschlüsse auf den Grund zuläßt. Hierdurch kann die spätere Fehlersuche erleichtert werden.

Kategorie	Grund	Folgeaktion	Art
1	Hardware defekt z.B. Tasten, Speicher	Abbruch des Programmlaufes	schwerer, nicht behebbarer Fehler
2	Fehler bei Dateneingabe oder Übertragung	Korrektur durch Aufforderung zur nochmaligen Eingabe oder Übertragung	behebbarer Fehler, Korrektur möglich. Eventueller Leistungsumfang eingeschränkt.
3	Auftreten von Ereignissen, z.B. Drucker nicht eingeschaltet	Meldung des Ereignisses in geeigneter Form	Fehler ohne weitreichende Bedeutung

Abb. 6.1 Kategorietabelle

Fehler der Kategorie 2 lassen sich durch eine Folgeaktion beheben und gewährleisten eine Aufrechterhaltung des Programmlaufs. Hierbei kann es sich um Benutzerfehler, z. B. die Eingabe eines falschen Kommandos oder um fehlerhafte Information, die z. B. von einer Datenübertragung herrührt, handeln.

Enthält die Information Übertragungsfehler, die im einfachsten Falle durch eine Paritätsprüfung erkannt wurden, so läßt sich der zugehörige Sender dazu stimulieren, die Sendung zu wiederholen. Treten beim nächstenmal immer noch Fehler auf, so wird eine weitere Wiederholung eingeleitet.

Um die Terminierung dieser Wiederholungen zu gewährleisten, muß ein Zähler mitlaufen, mit dessen Hilfe nach der n-ten Wiederholung dieser Zyklus abgebrochen wird. Dies kann unter Umständen in der Fehlerkategorie 1 enden, da dann offensichtlich die Übertragungseinheit defekt ist. Es kann jedoch auch dazu führen, daß das spezielle Gerät lediglich aus dem Verarbeitungsprozeß herausgenommen wird; z. B. wenn im Mailbox-Rechner ein Modem ausfällt. Der Betrieb kann dann mit nur drei Modems weiterlaufen.

Die Kategorie 3 behandelt Fehler oder Ereignisse, die keine signifikante Wirkung auf die Verarbeitungsabfolge haben. So wäre z. B. ein nicht angeschlossener Drucker, der durch eine Task angesprochen wird, der Verursacher eines solchen Fehlers. Die Folgeaktion ist trivial, da der Bediener lediglich aufgefordert werden muß, den Drucker einzuschalten.

6.3 Fehlerbehandlung von BSØ

Die Fehlerbehandlung von BSØ beschränkt sich nur auf Fehler, die beim Aufruf einer Kernprozedur auftreten können. Hierbei handelt es sich um formale Fehler, z. B. Senden

auf einen nicht konfigurierten Ringpuffer oder ähnliches. Die Kernprozeduren signalisieren einen Fehler dadurch, daß bei der Rückkehr das Carry Bit gesetzt ist. Der Inhalt des Akkus läßt weitere Schlüsse auf die Fehlerursache zu. Genauere Deteils sind dem Quellkode zu entnehmen, der für jede Prozedur eine Beschreibung enthält.

Andere Fehlerbehandlung, z. B. Übertragungsfehler etc., sind nicht Gegenstand des Systemkerns, sondern hängen stark von der jeweiligen Anwendung ab.

Zusammenfassung

In jedem realen Rechnersystem können Fehler auftreten. Die Ursachen dafür sind sehr vielschichtig und im System verstreut. Um dennoch die Gründe zusammenzufassen, teilt man sie in Kategorien auf und erhält so eine gewisse Struktur, die es erlaubt, auf gleichartige Fehler gleichartig zu reagieren.

Um den Betrieb eines Rechnersystems nicht durch einen gravierenden Fehler zu gefährden, strebt man generell eine Selbstkorrektur desselben an. Das kommt der Stabilität und damit der Qualität des Produktes zugute. Die Gestalt der Korrektur hängt dabei stark vom Problem selbst ab. Sie wird von einem Multitask-Kern unterstützt, der z. B. eine Wiederholung von Vorgängen in bestimmten Zeitintervallen einfach bewerkstelligen läßt.

Die anderen Fehlertypen führen entweder zum Exodus des Programms (Kategorie 1) oder produzieren lediglich eine Meldung oder Warnung (Kategorie 3).

7 Speicherverwaltung

Wird von Tasks ein gewisses Arbeitsspeichervolumen gebraucht, und muß dieses nicht ständig der Task zur Verfügung stehen, so wird die Einführung einer dynamischen Speicherverwaltung sinnvoll. Das Thema wurde bereits im Kapitel 4.4 angesprochen, wo eine solche Verwaltung im Zusammenhang mit Nachrichtenpuffern eingesetzt wurde. Dort wurden Pufferelemente dynamisch bei Bedarf angefordert und nach Gebrauch an die Speicherverwaltung zurückgegeben.

Diese Methode, denselben Speicherbereich mehreren Prozessen, wenn auch zeitlich versetzt, zugänglich zu machen, kann auch für die Lösung anderer Aufgaben Anwendung finden. Zum Beispiel ist es bei Sortiervorgängen immer gut, auf einen großen Arbeitsspeicher zurückgreifen zu können; dadurch wird der Vorgang wesentlich beschleunigt. Dies ist ein bedeutendes Kriterium beim Sortieren. Dagegen möchte man vermeiden, daß ein relativ großer Arbeitsspeicher einer einzigen Task exklusiv gehört.

Eine dynamische Speicherverwaltung besteht zum einen aus zwei Prozeduren, die den Bedarf anfordern bzw. den Bedarf abmelden (Speicher zurückgeben) und zum anderen aus dem Speicherbereich selber. Dieser Speichervorrat wird als Pool bezeichnet und logisch in gleichgroße Teile (Blöcke) aufgeteilt. Ein Block stellt somit für die Speicherverwaltung die kleinste zuteilbare Einheit dar. Typische Größen für einen Block sind 128, 256 oder 512 Bytes.

Das bedeutet, auch wenn ein Prozeß nur 1 Byte benötigt, bekommt er immer einen ganzen Block (z. B. 256 Bytes) zugewiesen.

Eine Task meldet ihren Wunsch nach Speicher über die Prozedur „allocate" an. Als Parameter übergibt sie die Anzahl der gewünschten Blöcke und bekommt von „allocate" einen Zeiger (Adresse) auf den zugeteilten Speicher als Ergebnis zurück. Ab dieser physikalischen Adresse hat nun die Task durchgehend Zugriff auf den Speicher in angefordertem Umfang.

Ist der Wunsch nach Speicher nicht erfüllbar, so erfolgt entweder eine Verzögerung durch eine Semaphore oder eine Fehlerrückmeldung an die aufrufende Task.

Die Speicherzuteilung erfolgt, wie erwähnt, blockweise. Dazu existiert eine Liste (Map) von Bits, die anzeigen, welche Blöcke belegt sind und welche nicht. Anhand dieser Bitmap stellt „allocate" fest, welcher Bereich frei ist und somit an eine anfordernde Task zugeteilt werden kann. *Abbildung 7.1* zeigt ein Beispiel.

Der *Speicherpool* besteht hier nur aus 10 Blöcken und beginnt bei Adresse 1000. Die Größe eines Blocks ist 256 Bytes (100 hex). MAP gibt an, welche Blöcke frei und welche belegt sind. Hier ist Block 1, 3, 4, 5, 8, 9 frei, der Rest ist durch andere Prozesse belegt.

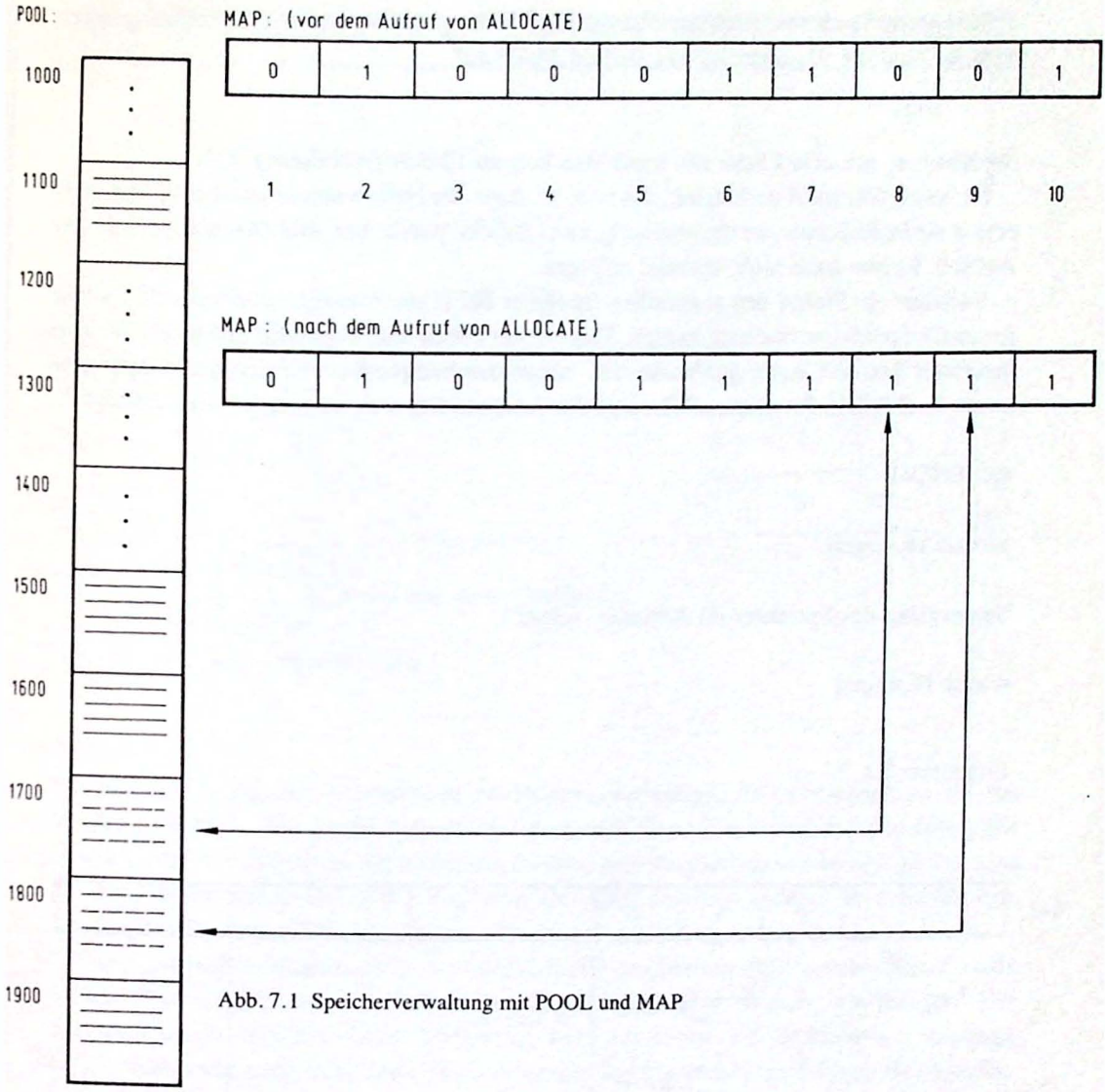


Abb. 7.1 Speicherverwaltung mit POOL und MAP

Wird „allocate" von einer Task mit dem Bedarf von 2 Blöcken aufgerufen, so wäre es am günstigsten, Block 8 und 9 zu vergeben. Als Zeiger wird die korrespondierende Adresse 1700 hex zurückgemeldet. Dieser Zuteilungsalgorithmus, der die Lücken aussucht, die optimal zum angemeldeten Bedarf passen, ist anhand der Bitmap relativ einfach zu erstellen.

Werden zwei freie und zusammenhängende Blöcke gesucht, wie es im Beispiel gefordert wird, so muß der Algorithmus das Bitfeld MAP auf

1001

durchsuchen, um eine Lücke mit zwei Blöcken zu finden *(Abbildung 7.2)*.

Ist diese Lücke nicht vorhanden, das ist z. B. dann der Fall, wenn alles frei ist, dann sucht er sich die Stelle heraus, an der erstmalig zwei Blöcke plazierbar sind *(Abbildung 7.3)*. Die Auswahl konnte dann nicht optimal erfolgen.

Nachdem ein Prozeß den zugeteilten Speicher für seine Zwecke verwendet hat, gibt er ihn an die Speicherverwaltung zurück. Das erfolgt durch eine Prozedur „release", die zwei Parameter hat; die Anfangsadresse des zusammenhängenden Adreßraumes und seine Länge in Blöcken. Programm 7.1 zeigt die Anwendung von „allocate" und „release".

```
task USER1
...
allocate (6,zeiger)
...
Verwendung des Speichers ab Adresse „zeiger"
...
release (6,zeiger)
...
```

Programm 7.1

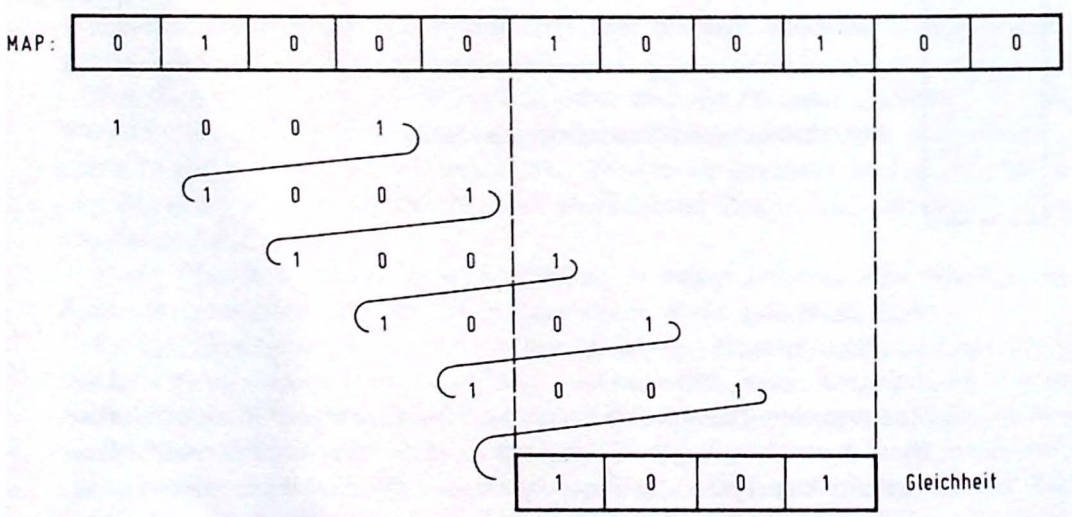


Abb. 7.2 Plazierung von zwei Blöcken

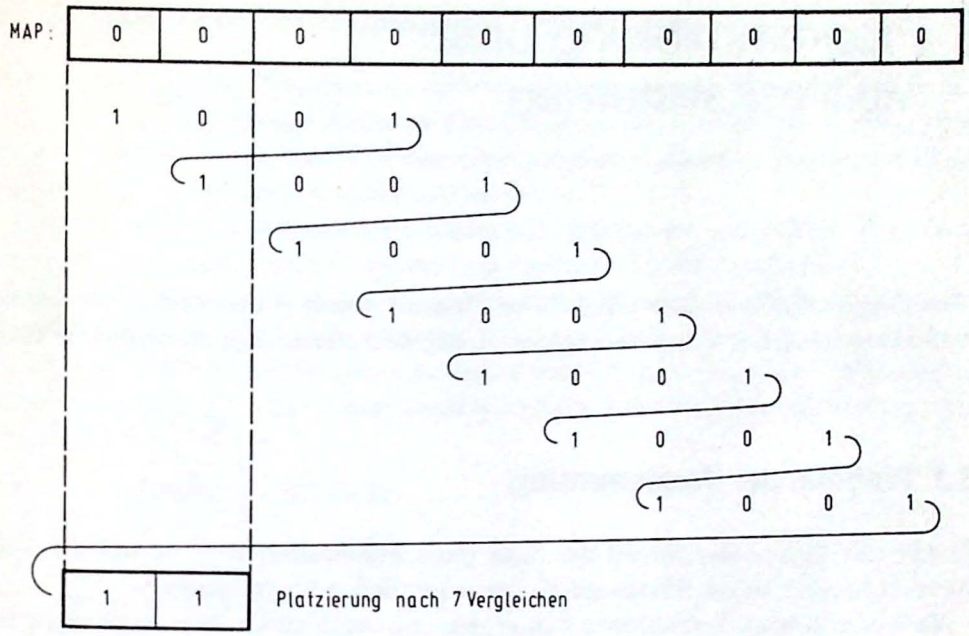


Abb. 7.3 Plazierung bei freiem Speicher

Programm 7.1 zeigt die Anwendung der Speicherverwaltung im Zusammenhang mit dem Mailbox-Rechner. Der große Arbeitsspeicherbedarf kann hier anfallen, wenn eine große Datenmenge eines Mailbox-Benutzers in die Mailbox eingetragen werden soll. Das ist dann der Fall, wenn ganze Dateien von einem Benutzer an einen anderen über die Mailbox übertragen werden sollen (Austausch von Texten oder Programmquellen).

Die empfangene Datei muß in der USER Task zwischengepuffert werden, bis sie weiter an den Box Handler gereicht wird. Dazu ist die Speicherverwaltung gut geeignet. Der Speicherbereich, der eine Datei aufnimmt, wird nur selten und nur kurzzeitig von einer USER Task gebraucht, demnach ist es sinnvoll, den Speicherbereich dynamisch anzufordern und freizugeben.

Zusammenfassung

Eine dynamische Speicherverwaltung besteht aus einem Speicherbereich, aus dem eine anfordernde Task über die Prozedur „allocate" bedient wird.

Durch „allocate" erlangt somit eine Task zusammenhängenden Speicherplatz, über den sie frei verfügen kann. Hat die Task die Arbeiten, für den sie den Speicher benötigte, erledigt, gibt sie ihn zur weiteren Disposition an die Verwaltung zurück.

8 Programmierung unter Multitasksystemen

Dieses Kapitel enthält einige Punkte, die im Zusammenhang mit paralleler Programmierung interessant sind, jedoch einzeln betrachtet, hier kein eigenes Kapitel ergeben würden.

8.1 Problem der Verklemmung

Erstellt man Gesamtlösungen auf der Basis eines Multitasksystems, so treten bei der Entwicklung ganz andere Effekte auf als bei sequentiellen Programmen.

Zwei oder mehrere Tasks können sich gegenseitig verklemmen, was letztendlich den Programmlauf blockiert.

Eine Verklemmung kann beispielsweise dann auftreten, wenn eine Semaphore falsch initialisiert wurde. Wird eine Semaphore mit Null initialisiert, und soll sie den Zugriff auf einen Drucker regeln (Beispiel Kapitel 3.1), so würde jeder Prozeß, der den Drucker verwenden will, verzögert werden. Kein einziger Prozeß könnte jemals drucken.

Eine Verklemmung dieser Art ist noch relativ offensichtlich. Tückischer wird es, wenn die Verklemmung nicht – wie hier – statisch, sondern dynamisch und damit abhängig von den vorherigen Aktionen auftritt.

Die Verklemmung zeigt sich dann meistens zu zufälligen Zeitpunkten und ist schwer zu rekonstruieren.

Bei einer Verklemmung spielen meistens folgende Punkte eine Rolle:

- Priorität einer Task
- Startzeitpunkt einer Task
- Abhängigkeit von Interrupts
- Fehler durch Initialisierung

Diese Punkte sind bei der Konstruktion von parallelen Programmen hinreichend zu prüfen, was durch entsprechende Tests und eingehende Kodeprüfung erfolgen kann.

8.2 Programmieren reentranter Prozeduren

Die Abhandlung von Teilaufgaben wird in Mehrprogrammsystemen vornehmlich von einzelnen Prozessen erledigt. Interessant jedoch ist auch die Struktur des Unterprogramms in herkömmlicher Weise. Dabei müssen Unterprogramme jedoch einigen Randbedingungen genügen, sie müssen nämlich reentrant sein.

Ein reentrantes Unterprogramm zeichnet sich dadurch aus, daß mehrere Prozesse es gleichzeitig verwenden können, es aber nur einmal im Speicher vorhanden ist.

Der Mailbox-Rechner verfügt über einen zentralen Kommandointerpreter, der von allen vier USER Tasks verwendet werden kann *(Abbildung 8.1)*. Er schlüsselt die Anweisungen, die von einem Endbenutzer gesendet werden, auf und überprüft sie auf formale Richtigkeit. Dieser Kommandointerpreter leitet anschließend die entsprechenden Folgeaktionen ein. Die Anweisung

MESSAGE TO FRITZ

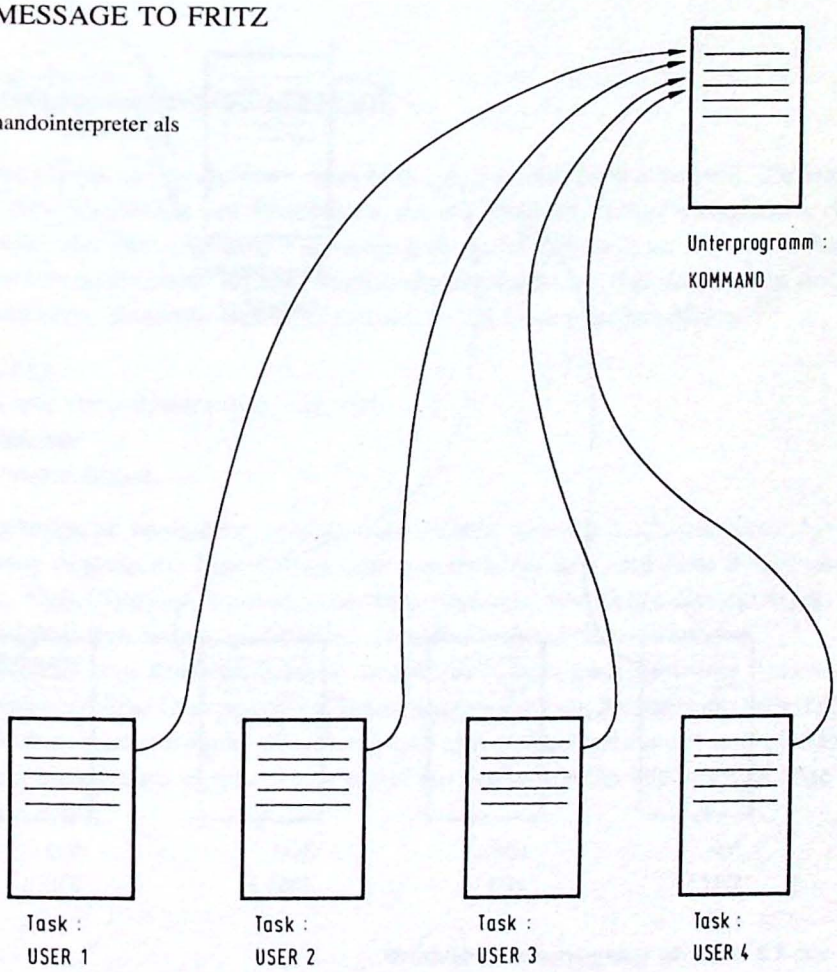


Abb. 8.1 Kommandointerpreter als Unterprogramm

(dabei ist FRITZ ein Mailbox Endbenutzer) würde zunächst daraufhin überprüft werden, ob sie richtig geschrieben und ob das Kommando MESSAGE zulässig ist etc. Danach wird das Versenden eines elektronischen Briefs eingeleitet.

Das Unterprogramm, das die Interpretation abhandelt, möge KOMMAND heißen. Es ist nun folgender Hergang denkbar:

USER1 ruft KOMMAND auf und beginnt die Interpretation einer Anweisung. Der Zeit-Interrupt schlägt mitten im KOMMAND zu und verzögert USER1 (Abbildung 8.2). Daraufhin wird vom Dispatcher die Task USER3 als nächste fortgeführt. Auch sie ruft KOMMAND auf. Sie durchläuft nun denselben Kodebereich wie USER1!

Bei Unterprogrammen dieser Art ist es deshalb wichtig, daß sie nicht mit absoluten Datenspeichern arbeiten. Man denke an die Folgen, wenn KOMMAND sich sein aktuell zu bearbeitendes Kommando (hier: MESSAGE TO FRITZ) in einer normalen Stringvariablen halten würde.

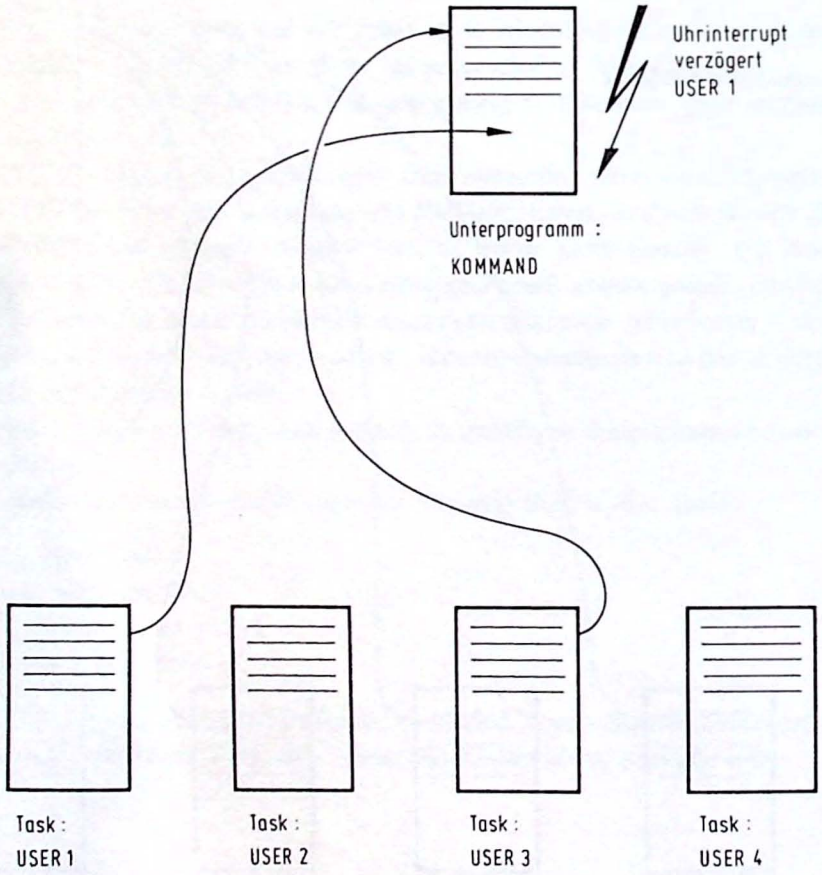


Abb. 8.2 Aufruf des Unterprogramms KOMMAND

USER3 würde so postwendend das noch nicht abgeschlossene Kommando von USER1 überschreiben, was zwangsläufig fatale Folgen haben würde.

Benötigt ein reentrantes Unterprogramm Speicher, so kann es sich diesen durch eine Speicherverwaltung besorgen, oder es verwendet den Stack. In unserem Falle würde KOMMAND das aktuelle Kommando auf den Stack schieben und dort mit ihm arbeiten.

Das ist deshalb zulässig, weil jede Task ihren eigenen Stackbereich besitzt; die Tasks kommen sich dadurch nicht ins Gehege.

Arbeitet man in einer Hochsprache, so ist genau zu prüfen, ob die Sprache reentranten Code erzeugt. Dies ist immer dann der Fall, wenn rekursive Prozeduren in einer Hochsprache erlaubt sind.

Rekursiv aufrufbare Unterprogramme (ein Unterprogramm ruft sich selber auf), müssen den genannten Bedingungen genügen, denn auch sie sind reentrant.

8.3 Unterprogrammbibliotheken

In den meisten Fällen ist das Anlegen einer Unterprogrammbibliothek sinnvoll. Darunter versteht man eine Sammlung von Prozeduren, die ein spezielles Fachgebiet umfassen. Es hängt dabei sehr von der einzelnen Anwendung ab, welche Prozeduren in einer solchen Bibliothek aufzunehmen sind. Wichtig bei Multitasksystemen ist, daß sie reentrant sind.

Es empfiehlt sich, folgende Bereiche generell in ein System zu integrieren:

- Stringhandling
- Mathematische Operationen (log, sin, cos, ...)
- BCD-Arithmetik
- Konvertierungsroutinen

Wird eine Hochsprache verwendet, so sind diese Gebiete meistens durch eine vorhandene Runtime Library abgedeckt. Hier jedoch muß gewährleistet sein, daß diese Prozeduren reentrant sind. Viele Compiler, besonders bei Mikrorechnern, sind nicht unter dem Aspekt: Multitask-Fähigkeit konzipiert, und können deshalb Probleme mit sich bringen.

Wenn im Notfall eine Runtime Library modifiziert werden muß, kann eine Prozedur reentrant gemacht werden. Dies geschieht, indem man am Eingang den Interrupt sperrt (DI) und vor dem Return wieder freigibt (EI). Dabei muß aber die Laufzeit einer Prozedur mit in die Überlegung einbezogen werden, da während des Prozedurlaufes kein Interrupt abgehandelt werden kann.

8.4 Hintergrundmonitor

Das Wort Monitor ist im Bereich der EDV ein vielbeanspruchter Begriff. Es wird für viele völlig verschiedene Dinge gebraucht. Unter anderem bezeichnet es ein maschinennahes Grundprogramm für Singletasksysteme. Es ist oft in Form eines EPROMs oder als zuladbares Teil bei der Grundausstattung eines Entwicklungssystems dabei.

Vorteilhaft ist es, wenn sich ein Monitor mit in ein Multitasksystem einbinden läßt. Er stellt dann eine Task dar, die mit geringer Priorität und somit im Hintergrund läuft. Der große Vorteil im Gegensatz zu Singletasksystemen ist der, daß nun parallel zum Programmablauf das System beobachtet werden kann.

Besonders deutlich wird der Vorteil beim Austesten von Realzeitanwendungen. Dabei läuft die eigentliche Anwendung im Vordergrund; der Monitor, über den das Geschehen beeinflußt werden kann, agiert nebenher im Hintergrund.

Der Monitor wird dann – von außen her – über ein eingenes Terminal bedient.

Die Verwendung eines solchen Monitors ist nicht für den Endbenutzer oder das Bedienungspersonal relevant, sondern lediglich für den Service oder zu Testzwecken.

Ein Monitor stellt allgemein folgende Kommandos zur Verfügung:

DUMP	Anzeigen eins konkreten Speicherinhaltes in hexademizaler Form und als ASCII-Zeichen.
REGISTER	Anzeigen der aktuellen CPU Registerstände.
I/0	Ein und Ausgabe von Bytes auf spezielle I/0 Ports.
MOVE	Verschieben eines Speicherbereiches von einer Adresse zu einer anderen.
FIND	Suchen von Byte-Folgen in einem angegebenen Speicherbereich.
ARITHM	Ausführen von hexadezimalen Grundrechenoperationen.
INSERT	Eintragen einer Folge von Bytes in konkrete Arbeitsspeicherzellen.
GO	Ausführen eines angegebenen Kodebereiches.
BREAK	Setzen von Breakpoints (Haltepunkten).
TRACE	Ausführen eines Kodebereiches mit Protokollierung jeden Befehls.

Weitere sehr nützliche, jedoch speicherintensive Kommandos sind:

ASS	Eingabe von Assemblerkode in mnemonischer Form.
DISASS	Disassemblieren (Rückassemblieren) eines bestimmten Speicherbereiches.

Die beschriebenen Funktionen laufen völlig isoliert vom Multitask-Kern und verwenden seine Dienste nicht. Deshalb ist es möglich, einen vorhandenen Monitor eines Singletasksystems in ein Multitasksystem einzubinden.

Ein solcher vorhandener Monitor sollte noch nach Möglichkeit um folgende Funktionen, die in einer Multitaskumgebung wichtig sind, erweitert werden.

SEMAPHORE	Anzeigen des Zustandes einer Semaphore. Dazu gehört Zählerstand sowie evtl. in ihr wartende Prozesse. Im gleichen Kommando sollte eine Semaphore auf einen konkreten Wert gesetzt werden können.
TASKS	Anzeigen der existierenden Tasks, mit Status und Zeitzyklusangaben, Prioritäten usw.
STATUS	Anzeigen und Manipulation eines Taskstatus. Hiermit kann eine Task expliziert, z. B. von „terminated" nach „ready" gesetzt werden.
SCHEDUL	zeigen und Verändern von Schedule-Parametern (Zykluszeit etc.).
BUFFER	Information über Nachrichtenpuffer. Ausgabe von interessanten Daten über einen Puffer; dazu gehört: Anzahl der gepufferten Elemente und deren Inhalt.

Anhang

BS0 Listings mit Beispieltasks.

```
Systemkern BS0          (180685)      MACRO-80 3.43   27-Jul-81       PAGE    1

                        ;
                        ;
                        ;
                        ;     ****      ****      ****
                        ;     *  *    *         *  * *
                        ;     ****      ****     * *  *
                        ;     *  *         *    **   *
                        ;     ****      ****      ****
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; Betriebssystem fuer Realzeitanwendungen auf dem Z80
                        ;
                        ;*****************************************************************
                        ;
                        ; Autor: Dipl. Inform. (FH) Michael Zilker
                        ;        Kirchschoenbach 39 a
                        ;        8718 Prichsenstadt
                        ;
                        ; Datum: August 84
                        ;
                        ; Zielprozessor: Z80
                        ;
                        ;
                        ;
                                TITLE  Systemkern BS0                   (180685)
                                SUBTTL  Scheduler / Dispatcher
                                .Z80
                                .SALL
                                PAGE
```

```
                EXT     BASIS,TREIB2,CRASH
                EXT     CLKINI,PIOINI
                PUBLIC ENDE,ACTIVE,DELAY,INISEM,WAIT,SIGNAL
                PUBLIC SIGFI,SEND,RECEIV
        ;
        ;
        ;*****************************************************************
        ;
        ; MACRO Definitionen zum Sichern und Restaurieren der Register sowie SP
        ;
        ;*****************************************************************
        ;
        ;
+       RSAVE   MACRO           ;Register auf dem aktuellen Stack sichern
+               PUSH    AF      ;Akku und Flags
+               PUSH    BC      ;Arbeitsregister
+               PUSH    DE
+               PUSH    HL
+               PUSH    IX      ;Sichern der Indexregister des Z80
+               PUSH    IY      ;In diesem MACRO muessten auch die Hintergrundregister
+                               ;gesichert werden. Dies entfiel hier, koennte aber ohne
+                               ;Problem noch eingefuegt werden.
+
+               LD      IX,(DESADR)   ;Uebergang running --> ready
+               LD      (IX+STATUS),READY
                ENDM
        ;
        ;
        ;
+       RRELOD  MACRO           ;Bringt die gesicherten Register wieder auf den alten Stand
+
+               LD      IX,(DESADR)   ; Uebergang ready --> running
+               LD      (IX+STATUS),RUNING
+
+               POP     IY
+               POP     IX      ;Hier muessten dann die Hintergrundreg. eingefuegt werden
+               POP     HL
+               POP     DE
+               POP     BC
+               POP     AF
                ENDM
        ;
        ;
        ;
+       SPSAVE  MACRO                   ;Sichert den Stackpointer in BS0 ueblicher Form
+                                       ;im Task Deskriptor Block der aktuell laufenden Task
+               LD      HL,0H           ;StackPointer ueber den ADD Befehl ins HL bringen
+               ADD     HL,SP           ;Direktes Laden beim Z80 nicht moeglich.
+               LD      DE,(DESADR)     ;Die Adresse des akt. Task Deskriptors steht in DESADR
+               EX      DE,HL           ; <DE> = zu sichernder SP
+               LD      (HL),E          ; <HL> = zeigt auf Stelle im TCB wo der SP
+               INC     HL              ; gesichert werden soll.
+               LD      (HL),D          ;
                ENDM
```

```
Systemkern BS0                (180685)        MACRO-80 3.43   27-Jul-81       PAGE    1-2
Scheduler / Dispatcher

                              ;
                              ;
                              ;
                              ;
                        +     SPRELO  MACRO                   ;Laedt den Stack Pointer der unterbrochenen
                        +             LD      HL,(DESADR)     ;Task zurueck.
                        +             LD      E,(HL)
                        +             INC     HL
                        +             LD      D,(HL)          ;jetzt Hi Byte
                        +             EX      DE,HL
                        +             LD      SP,HL           ;Stack pointer ladren
                                      ENDM

                              ;
                              ;
                                      PAGE

                              ;
                              ;
                              ;
                              ;*****************************************************************
                              ;
                              ; Datenbereich vom BS0 Kern
                              ;
                              ;*****************************************************************
                              ;
0000'                                 DSEG
0000'   0035'                 INTTAB: DW      TICK            ;Interrupt Tabelle muss auf einer neuen Seite beginnen
0002'   0132'                         DW      VECT2           ;Interrupt Vector 2
0004'   0171'                         DW      VECT4           ;    "       "    4
0006'   0171'                         DW      VECT6           ;    "       "    6
0008'   0171'                         DW      VECT8           ;    "       "    8
000A'   0171'                         DW      VECTA           ;    "       "    A
000C'   0171'                         DW      VECTC           ;    "       "    C
000E'   0171'                         DW      VECTE           ;    "       "    E
0010'   0171'                         DW      VECT10          ;    "       "    10
                              ;
                              ;
0012'                                 DS      50D             ;Betriebssystem Stack
0044'   00                    STACK:  DB      0H
                              ;
0045'   0044'                 SYSSTK: DW      STACK           ;Hier wird der StackPointer vom BS0 Kern abgelegt
0047'                                 DS      10D             ;Dummy TCB fuer IDLE TASK
                              ;
0051'   0045'                 DESADR: DW      SYSSTK          ;Task Deskriptor Adresse der augenblicklick laufenden
                                                              ;Task. Am Anfang laeuft die IDLE Task mit dem
                                                              ;System Stack.
                              ;
                              ;
                              ;
                                      PAGE
```

```
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; EQUATES zum Betriebssytem
                        ;
                        ;*****************************************************************
                        ;
                        ;
                        ;
                        ; Hier die maximale Task und Semaphorenanzahl bei der Systemgenerierung
                        ; angeben
0005                    NTASKS  EQU     5H
0003                    NSEMAS  EQU     3H
0003                    NPUF    EQU     3H      ;Anzahl der Ringpuffer hier angeben
0004                    PUFLNG  EQU     4H      ;Anzahl der Byts pro Puffer Eintrag
0003                    MAXENT  EQU     3H      ;Maximale Anzahl der Eintraege pro Puffer
                        ;
                        ;
                        ;
                        ; Offsetwerte im Task Deskriptor Block
                        ;
0000                    STACKP  EQU     0D              ;Sichern des Task privaten Stack Pointers
0002                    STATUS  EQU     2D              ;Aktueller Prozesstaus
                                                        ;Running/Ready/Waiting/Terminated
0003                    START   EQU     3D              ;Start Adresse der Task
0005                    TIMER   EQU     5D              ;Zeit Zaehler (wird alle 10ms dekrementiert)
0006                    INTERV  EQU     6D              ;Task wird im Intervall gestartet
0007                    DLYTIM  EQU     7D              ;Timer falls Task verzoegert werden soll
0008                    NAME1   EQU     8D              ;Task Name 1. Byte
0009                    NAME2   EQU     9D              ;Task Name 2.Byte
                        ;
                        ;
                        ;
                        ; Prozesszustaende
                        ;
0001                    RUNING  EQU     00000001B       ;Rechnend (diesen Status hat immer nur eine Task)
0002                    READY   EQU     00000010B       ;Bereit zum Rechnen
0004                    WAITNG  EQU     00000100B       ;Task wartet auf Semaphore
0008                    TERMIN  EQU     00001000B       ;Task ist beendet - terminated
0010                    DLYED   EQU     00010000B       ;Task ist Time delayed
                        ;
                        ;
                        ; Returnstati aus Unterprogrammaufrufen von ACTIVE,...
                        ;
0000                    NOERR   EQU     0H              ;Alles klar, Returnstatus ok!
0010                    RSNOTS  EQU     10H             ;Task# nicht vorhanden
0011                    RSTSWA  EQU     11H             ;Task kann nicht aktiviert werden, denn sie ist
                                                        ;im Zustand waiting oder delayed.
0020                    RSNOSE  EQU     20H             ;Sema# nicht vorhanden.

0030                    RSNOBU  EQU     30H             ;Puffer Nummer nicht konfiguriert

                                PAGE
```

```
Systemkern BS0                  (180685)        MACRO-80 3.43   27-Jul-81       PAGE    1-5
Scheduler / Dispatcher


0053*                                   CSEG
                                ;
                                ;
                                ;*****************************************************************
                                ;
                                ; Power Up und Idle Part
                                ;
                                ;*****************************************************************
                                ;
0000'   F3                      POWUP:  DI                      ;Interrupt zunaechst sperren
0001'   ED 5E                           IM      2               ;Interrupt Modus 2
0003'   21 0000*                        LD      HL,INTTAB       ;I Register setzen
0006'   7C                              LD      A,H
0007'   ED 47                           LD      I,A
0009'   31 0044*                        LD      SP,STACK        ;Stack Pointer auf SystemStack setzen
000C'   CD 03DE'                        CALL    INIPUF          ;Semas der Pufferverwaltung initial.
000F'   CD 0000#                        CALL    CLKINI          ;Timer Baustein initialisieren
0012'   CD 0000#                        CALL    PIOINI          ;PIO Initialisieren
0015'   FB                              EI                      ;Interrupt freigeben
                                ;
0016'   76                      IDLE:   HALT                    ;Idle Task
0017'   C3 0016'                        JP      IDLE
                                ;
                                ;
                                        PAGE
```

```
                        ;
                        ;*****************************************************************
                        ;
                        ; Programmkode zum BS Kern
                        ;
                        ;*****************************************************************
                        ;
                        ;
                        ;
                        ; Zunaechst einige Unterprogramme die der Kern intern verwendet.
                        ;
001A'                   DURETI:                 ;Dummy Return from Interrupt
001A'   ED 4D                   RETI            ;Z80 Eigenart
                                                ;Meldet an Timer Baustein das Ende der Interrupt Service Rout.
                        ;
                        ;
001C'   DD 7E 02        SETRDY: LD      A,(IX+STATUS)           ;Ready setzen einer Task mit Pruefung ob waiting
001F'   FE 04                   CP      WAITNG                  ;Abrage ob Task im Zustand Wartend
0021'   CA 0032'                JP      Z,SETERR                ;falls ja dann Fehlerreturn.
0024'   FE 10                   CP      DLYED                   ;Abfrage ob Task delayed
0026'   CA 0032'                JP      Z,SETERR                ;falls ja dann Fehlerreturn.
0029'   DD 36 02 02             LD      (IX+STATUS),READY       ;Ready setzen im Deskr. Block
002D'   3E 00                   LD      A,NOERR                 ;Return Status ok!
002F'   C3 0034'                JP      STREND                  ;goto ende
0032'   3E 11           SETERR: LD      A,RSTSWA                ;Return Status : Task ist wartend,delayed
0034'   C9              STREND: RET                             ;RETURN SETRDY
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; TICK: hier schlaegt der Timer Interrupt alle 10ms ein.
                        ;
                        ;*****************************************************************
                        ;
0035'   F3              TICK:   DI


                  +             RSAVE                   ;Register und Stack Pointer sichern
                  +             SPSAVE


0052'   ED 7B 0045*             LD      SP,(SYSSTK)     ;System Stack Pointer laden
                        ;
                        ;
                        ; Hier muessen alle Task Deskriptoren Blocke abgearbeitet werden.
                        ; Die Timer dekrementieren und Die Task ready setzen falls der Timer abgelaufen.
                        ;
0056'   FD 21 0001*             LD      IY,BASIS+1      ;B zaehlt die Tasks
005A'   3A 0000*                LD      A,(BASIS)       ;IY zeigt auf die Adresse des gerade bearbeiteten
005D'   47                      LD      B,A             ;Task Desk. Block.
                        ;
```

```
005E'  FD 5E 00        AGAIN:  LD      E,(IY)          ;IX wird auf Task Desk. Block gelegt
0061'  FD 56 01                LD      D,(IY+1)
0064'  D5                      PUSH    DE
0065'  DD E1                   POP     IX              ;  LD   IX,(IY) !!!!
                       ;
0067'  DD 35 05                DEC     (IX+TIMER)              ;Timer = Timer -1
006A'  C2 0070'                JP      NZ,CHKINT               ;if TIMER <> 0 then goto CHKINT
006D'  CD 001C'                CALL    SETRDY                  ;else set Task ready
                       ;
0070'  3E FF           CHKINT: LD      A,0FFH                  ;if Timer <> 0FFH then goto NEXTSK
0072'  DD BE 05                CP      (IX+TIMER)
0075'  C2 007E'                JP      NZ,DLYCNT
0078'  DD 7E 06                LD      A,(IX+INTERV)           ;                       else refresh Timer
007B'  DD 77 05                LD      (IX+TIMER),A
                       ;
                       ;
                       ; Verarbeitung des Zaehlers falls Task delayed wurde.
                       ;
                       ;
007E'  3E 00           DLYCNT: LD      A,0H                    ;Nachsehen ob Delay Counter <> 0
0080'  DD BE 07                CP      (IX+DLYTIM)
0083'  CA 0090'                JP      Z,NEXTSK                ;Falls Zelle = Null dann weiter

0086'  DD 35 07                DEC     (IX+DLYTIM)             ;sonst Counter dekrementieren
0089'  C2 0090'                JP      NZ,NEXTSK               ;und bei nullwerden der Zelle die
008C'  DD 36 02 02             LD      (IX+STATUS),READY       ;Task ready setzen. SETRDY darf nicht
                                                               ;aufgerufen werden, da es wartende Tasks
                                                               ;nicht ready setzt!

0090'                  NEXTSK:                                 ;Steht noch eine Task zur Verarbeitung an
0090'  FD 23                   INC     IY                      ;ja : IY neu positionieren
0092'  FD 23                   INC     IY
0094'  10 C8                   DJNZ    AGAIN                   ;zurueck zur naechsten Task
                       ;
                       ;
                       ;
0096'  CD 001A'        TSKEND: CALL    DURETI                  ;Timer freigeben
                                                               ;Trotzdem Interrupt sperren
                       ;
                       ;
                       ;-----------------------------------------------------------------------
                       ; SELTSK : Select Task - naechste hoechstpriorisierte Task auswaehlen und starten
                       ;-----------------------------------------------------------------------
                       ;
0099'  FD 21 0001*     SELTSK: LD      IY,BASIS+1              ;Task mit hoechster Prioritaet (erste
009D'  3A 0000*                LD      A,(BASIS)               ;Task in der Kette die ready ist) suchen
00A0'  47                      LD      B,A                     ;B zaehlt die Tasks runter
00A1'  3E 02                   LD      A,READY                 ;READY Maske im Akku
                       ;
00A3'  FD 5E 00        NEXTCB: LD      E,(IY)                  ;IX soll auf Desk. Block zeigen
00A6'  FD 56 01                LD      D,(IY+1)
00A9'  D5                      PUSH    DE
00AA'  DD E1                   POP     IX                      ;jetzt tut er es.
                       ;
00AC'  DD BE 02                CP      (IX+STATUS)             ;Status mit Ready vergleichen
00AF'  CA 00D4'                JP      Z,RUNTSK                ;Task gefunden, die ready ist
```

```
                                                        ; dann weiter bei RUNTSK
00B2'  FD 23                    INC     IY              ;sonst naechste Task ansehen
00B4'  FD 23                    INC     IY
00B6'  10 EB                    DJNZ    NEXTCB          ;Falls noch eine Task ansteht dann NEXTCB
                                                        ; sonst IDLE Task fortfuehren.
00B8'  21 0045*         RUNIDL: LD      HL,SYSSTK       ;DESADR eintragen hier SYSSTK
00BB'  22 0051*                 LD      (DESADR),HL
                        ;
                +               RRELOD                  ;Register zurueck laden
00CE'  ED 73 0045*              LD      (SYSSTK),SP     ;und SP sichern
00D2'  FB                       EI                      ;Int. scharf machen und IDLE starten
00D3'  C9                       RET
                        ;
                        ;
                        ;
00D4'                   RUNTSK:                         ;Status eintragen

00D4'  ED 73 0045*              LD      (SYSSTK),SP     ;System Stack Pointer sichern
00D8'  DD 6E 00                 LD      L,(IX)          ;alten Stack Pointer heraus kramen
00DB'  DD 66 01                 LD      H,(IX+1)
00DE'  F9                       LD      SP,HL           ;und nach echtem SP bringen
                        ;
00DF'  DD 22 0051*              LD      (DESADR),IX     ;Aktuelle Deskriptoren Adresse sichern
                        ;
                +               RRELOD                  ;Alte Registerstaende holen
00F3'  FB                       EI
00F4'  C9                       RET                     ;und Task starten bzw fortfuehren
                        ;
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; ENDE: hier verabschiedet sich eine Task, sie geht in den Status TERMINATED
                        ;
                        ;*****************************************************************
                        ;
00F5'  F3               ENDE:   DI                      ;Interrupt entschaerfen
00F6'  E5                       PUSH    HL              ;Dummy Push spaeter wird hier Startad. eingetragen
                +               RSAVE                   ;privaten Stack richtigstellen
                +               SPSAVE                  ;Stack Pointer sichern. Task regulaer
                                                        ;abschliessen.
0113'  DD 2A 0051*              LD      IX,(DESADR)     ;Desk. Block verfuegbar machen
0117'  DD 2B                    DEC     IX              ;IX=IX-2
0119'  DD 2B                    DEC     IX              ;IX zeigt jetzt auf Program Counter Platz
                                                        ;im Desk. Block. Dort muss die Start Adresse
                                                        ;wieder restauriert werden.
011B'  DD 6E 05                 LD      L,(IX+START+2)  ;HL=Start Adresse
011E'  DD 66 06                 LD      H,(IX+START+3)  ;
0121'  DD 75 00                 LD      (IX),L
0124'  DD 74 01                 LD      (IX+1),H

                        ;
0127'  DD 36 04 08              LD      (IX+STATUS+2),TERMIN    ;Status terminated eintragen
012B'  ED 7B 0045*              LD      SP,(SYSSTK)     ;System Stack holen
012F'  C3 0096'                 JP      TSKEND          ;und weiter bei TSKEND
                        ;
                        ;
                        ;
                        ;
                                PAGE
```

```
Systemkern BS0                  (180685)        MACRO-80 3.43   27-Jul-81       PAGE    1-10
Scheduler / Dispatcher


                                ;
                                ;
                                ;
                                ;*****************************************************************
                                ;
                                ; FLIH - FIRST LEVEL INTERRUPT HANDLER
                                ; Interrupt Verarbeitung   Interrupthandler
                                ;
                                ;*****************************************************************
                                ;
                                ;
0132'   F3                      VECT2:  DI                      ;Es folgen die Adressen, die bei einem bestimmten
                                                                ;Interrupt angesprungen werden.
                        +               RSAVE                   ;Register sichern
                        +               SPSAVE                  ;Stackpointer im Deskriptor Block sichern
014F'   ED 7B 0045*                     LD      SP,(SYSSTK)     ;Auf Systemstack gehen
0153'   CD 0000*                        CALL    TREIB2
                        +               SPRELO
                        +               RRELOD
016E'   FB                              EI
016F'   ED 4D                           RETI
                                ;
0171'                           VECT4:                          ;Folgende Teile werden erst bei Bedarf konfiguriert
0171'                           VECT6:
0171'                           VECT8:
0171'                           VECTA:
0171'                           VECTC:
0171'                           VECTE:
0171'   ED 4D                   VECT10: RETI
                                ;
                                ;
                                ;
                                ;
                                ;
                                        PAGE
```

```
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; SETA
                        ;
                        ;*****************************************************************
                        ;
                        ; Setzt den Akku im gesicherten Registerblock, dient zur Rueckmeldung
                        ; etwaiger Fehlerkodes
                        ;
0173'   E5              SETA:   PUSH    HL
0174'   C5                      PUSH    BC
0175'   D5                      PUSH    DE
0176'   2A 0051'                LD      HL,(DESADR)     ;Stack Pointer der zuletzt laufenden Task holen
0179'   5E                      LD      E,(HL)
017A'   23                      INC     HL
017B'   56                      LD      D,(HL)
017C'   EB                      EX      DE,HL
                                                        ;HL ist SP der letzten running Task
017D'   01 000B                 LD      BC,11D
0180'   09                      ADD     HL,BC
0181'   77                      LD      (HL),A          ;Wert reinschreiben
0182'   D1                      POP     DE
0183'   C1                      POP     BC
0184'   E1                      POP     HL
0185'   C9                      RET
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; DELAY
                        ;
                        ;*****************************************************************
                        ;
                        ; Die aufrufende Task wird um  (Akku) * 10ms verzoegert
                        ; Der Akku enthaelt somit die Verzoegerungszeit.
                        ;
0186'   F3              DELAY:  DI
                +               RSAVE                   ;Register und Stackpointer sichern
                +               SPSAVE
01A3'   ED 7B 0045'             LD      SP,(SYSSTK)     ;Auf System Stack umschalten
01A7'   DD 2A 0051'             LD      IX,(DESADR)     ;IX auf Deskr. Block legen
01AB'   DD 36 02 10             LD      (IX+STATUS),DLYED ;Status = delayed
01AF'   DD 77 07                LD      (IX+DLYTIM),A   ;AKKU enthaelt Vezoegerungszeit
01B2'   C3 0099'                JP      SELTSK          ;naechste Task starten.
                        ;
                        ;
                        ;
                        ;
                        ;
```

```
                        ;*****************************************************************
                        ;
                        ; ACTIVE
                        ;
                        ;*****************************************************************
                        ;
                        ; Durch Aufruf von ACTIVE wird eine Task von einer anderen Task aktiviert
                        ; (ready gesetzt).
                        ;
                        ; (BC) --> den Taskname 2 Char. (eigentlich beliebiger 16 Bit Wert)
                        ;
                        ; Der Reurnstatus wird im Akku zuruckuebergeben.
                        ;
01B5'  F3               ACTIVE: DI
                  +             RSAVE
                  +             SPSAVE                  ;Register etc. sichern
01D2'  ED 7B 0045*              LD      SP,(SYSSTK)     ;auf den Systemstack schalten

01D6'  C5                       PUSH    BC              ;Taskname ins DE schaffen
01D7'  D1                       POP     DE

01D8'  3A 0000*                 LD      A,(BASIS)       ;Task Anzahl im B herunterzaehlen
01DB'  47                       LD      B,A
01DC'  FD 21 0001*              LD      IY,BASIS+1

01E0'  FD 6E 00         ACTAG:  LD      L,(IY)          ;Adresse des b ten Tsk. Desk. bringen
01E3'  FD 66 01                 LD      H,(IY+1)
01E6'  E5                       PUSH    HL
01E7'  DD E1                    POP     IX              ;IX zeigt jetzt auf den TCB

01E9'  7B                       LD      A,E             ;nachsehen ob der Taskname gleich dem
01EA'  DD BE 08                 CP      (IX+NAME1)      ;gesuchten ist
01ED'  C2 0207'                 JP      NZ,CHKNEX       ;if <> dann naechsten TCB ansehen

01F0'  7A                       LD      A,D             ;Erstes Byte war gleich, deshalb zweites
01F1'  DD BE 09                 CP      (IX+NAME2)      ;Byte pruefen
01F4'  C2 0207'                 JP      NZ,CHKNEX

01F7'  CD 001C'                 CALL    SETRDY          ;Task gefunden, also ready setzen
01FA'  FE 00                    CP      NOERR           ;Trat Fehler auf?
01FC'  C2 020F'                 JP      NZ,ACTER

01FF'  3E 00                    LD      A,NOERR         ;Return Status = OK
0201'  CD 0173'                 CALL    SETA            ;im Registersatz vermerken
0204'  C3 0099'                 JP      SELTSK          ;weiter mit der naechsten Task

0207'  FD 23            CHKNEX: INC     IY              ;Auf naechsten TCB gehen
0209'  FD 23                    INC     IY
020B'  10 D3                    DJNZ    ACTAG           ;weiter beim naechsten TCB
020D'  3E 10                    LD      A,RSNOTS

020F'  CD 0173'         ACTER:  CALL    SETA            ;Faellt er hier durch, dann wurde
                  +             SPRELO                  ;ein falscher Name angegeben, also Fehler
                  +             RRELOD
022A'  FB                       EI
022B'  C9                       RET
```



```
                                PAGE
```

```
                        SUBTTL  Semaphoren Verwaltung
                ;
                ;
                ;
                ;
                ;*****************************************************************
                ;
                ; Semaphoren Verwaltung fuer BS0
                ;
                ;*****************************************************************
                ;
                ;
                ;
                ; Das Datensegment wird fuer NTASKS,NSEMAS (maximale. Anzahl der Tasks und
                ; der Semaphoren bei Systemgenerierung) angelegt. Es handelt sich hier um
                ; eine statische Datenstruktur. Sie wurde der Einfachheit und der Uebrsicht-
                ; lichkeit gewaehlt.
                ;
                ;
022C'                   DSEG
0053"           SEMAS:  DS      (NSEMAS+NPUF*2) * ( ( (NTASKS-1) * 2) + 3 )
00B6"                   CSEG
                ;
                ;
                ; Equates fuer Semaphoren Verwaltung
                ;
000B            SEMOFF  EQU     ( ( ( NTASKS - 1 ) * 2) + 3)            ;Weite von einer Sema
                                                                        ;zur anderen.
0000            COUNT   EQU     0H              ;Counter im Sema Block
0001            PUTIDX  EQU     1H              ;Index fuer Einklinken in die Warteschlange
0002            GETIDX  EQU     2H              ;Index fuer Ausloesen aus der Warteschlange
0003            SBAS    EQU     3H              ;Dort beginnt der Schlangen Bereich!
                ;
                ;
                ;
                ;
                        PAGE
```

```
                        ;
                        ;
                        ;-----------------------------------------------------------------------
                        ;
                        ; Unterprogramme, welche Semaphoren Verwaltungs intern verwendet werden
                        ;
                        ;-----------------------------------------------------------------------
                        ;
                        ;
                        ; IYPOS setzt den IY Pointer auf den Sema Block, dabei wird die Sema# im Akku
                        ;       uebergeben.
022C'   C5              IYPOS:  PUSH    BC              ;BC und DE sichern
022D'   D5                      PUSH    DE

022E'   47                      LD      B,A             ;Offset zu SEMAS berechnen
022F'   04                      INC     B               ;B=B+1

0230'   FD 21 0053"             LD      IY,SEMAS        ;Basisadresse SEMAS in IY
                                                        ;Berechne: IY=SEMAS+A*SEMOFF !
0234'   11 000B                 LD      DE,SEMOFF       ;DE fuer Addition mit SEMOFF laden
0237'   10 03           ML:     DJNZ    MLADD
0239'   C3 0241'                JP      MLEND           ;if B=0 then Ende der Berechnung
023C'   FD 19           MLADD:  ADD     IY,DE           ;Addiere einmal den Offset
023E'   C3 0237'                JP      ML
0241'   D1              MLEND:  POP     DE              ;IY zeigt jetzt auf gewuenschten Semaphoren
0242'   C1                      POP     BC              ;Block.
0243'   C9                      RET
                        ;
                        ;
                        ;
                        ; VALID ueberprueft ob die Sema# im gueltigen Bereich liegt, dabei wird
                        ;       die Nummer im Akku uebergeben.
                        ;       Nummer is gueltig im Bereich von 0..NSEMAS-1
                        ;
0244'   FE 09           VALID:  CP      NSEMAS+2*NPUF   ;If A<NSEMAS then set Carry
                                                        ;            else reset Carry
0246'   3F                      CCF
0247'   D0                      RET     NC              ;Return falls ok
0248'   3E 20                   LD      A,RSNOSE        ;Error Kode nach A Fehlerfall.
024A'   C9                      RET
                        ;
                        ;
                        ;
                        ; MOD berechnet den Rest einer Byte Division.
                        ;     A = A modulo B
                        ;
024B'   90              MOD:    SUB     B               ;einmal subtrahieren.
024C'   DA 0252'                JP      C,MODEND        ;if A<B then MODEND
024F'   C3 024B'                JP      MOD
0252'   80              MODEND: ADD     A,B             ;Rest im Akku aufbauen.
0253'   C9                      RET
                        ;
                                PAGE
```

```
                                ;
                                ;
                                ;*****************************************************************
                                ;
                                ; INISEM initialisiert eine Semaphorvariable mit einem konkreten Wert
                                ;
                                ;       Parameter:      Akku  muss Sema# enthalten
                                ;                       C Reg. muss Initialwert enthalten.
                                ;                       --> Akku Returnstatus ok=0
                                ;
                                ;*****************************************************************
                                ;
0254'   F3                      INISEM: DI                      ;Hier darf kein Interrupt einschlagen.
0255'   CD 0244'                        CALL    VALID           ;Ist die # Gueltig?
0258'   DA 0295'                        JP      C,BADRET
025B'   C3 025F'                        JP      DOINI

025E'   F3                      ININCH: DI                      ;Entry ohne gueltigkeitscheck


025F'                   +       DOINI:  RSAVE

026F'   FD E5                           PUSH    IY              ;IY sichern

0271'   CD 022C'                        CALL    IYPOS           ;IY auf entsprechenden Sema Block positionieren.
0274'   FD 71 00                        LD      (IY+COUNT),C    ;Counter der Sema initialisieren.
0277'   FD 36 01 00                     LD      (IY+PUTIDX),0H  ;Sowie die Indizes.
027B'   FD 36 02 00                     LD      (IY+GETIDX),0H

027F'   FD E1                   INIRET: POP     IY              ;IY restaurieren.

                        +               RRELOD
0291'   3E 00                           LD      A,NOERR         ;RS=ok
0293'   FB                              EI
0294'   C9                              RET

0295'                           BADRET:
0295'   3E 20                           LD      A,RSNOSE        ;Fehler zuruckmelden
0297'   FB                              EI
0298'   C9                              RET                     ;und in die Task zuruck kehren.

                                        PAGE
```

```
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; WAIT Funktion fuer Semaphoren.
                        ;
                        ;       Akku muss Sema# enthalten.
                        ;
                        ;*****************************************************************
                        ;
0299'   F3              WAIT:   DI
                +               RSAVE                   ;Register unbedingt sichern da diese Task
                +               SPSAVE                  ;evtl. verdraengt wird.
02B6'   ED 7B 0045"             LD      SP,(SYSSTK)     ;Auf System Stack uebergehen.

02BA'   CD 0244'                CALL    VALID           ;Ist Die Semaphore vorhanden?
02BD'   DA 030E'                JP      C,WERRET        ;falls nicht --> Error Return

02C0'   CD 022C'                CALL    IYPOS           ;IY auf Sema Block setzten

                                                        ;Nachsehen ob die Sema belegt ist
02C3'   3E 00                   LD      A,0H
02C5'   FD BE 00                CP      (IY+COUNT)      ;If Counter=0 dann ist Sema belegt
02C8'   C2 0303'                JP      NZ,FREI

02CB'   26 00           BELEGT: LD      H,0H            ;Aktuelle Task in die Warteschlange einreihen
02CD'   FD 6E 01                LD      L,(IY+PUTIDX)   ;und Task waiting setzen.
02D0'   29                      ADD     HL,HL           ;Index mal zwei nehmen.
02D1'   FD E5                   PUSH    IY
02D3'   DD E1                   POP     IX              ;IX=IY !!!!
02D5'   E5                      PUSH    HL
02D6'   D1                      POP     DE              ;DE=HL
02D7'   DD 19                   ADD     IX,DE           ;IX=IX+Index
02D9'   11 0003                 LD      DE,SBAS         ;SBAS gibt Offset zum "Uebersoringen" von
                                                        ;Counter und Index Elementen.
02DC'   DD 19                   ADD     IX,DE           ;IX zeigt jetzt auf Adresse wo aktuelle
                                                        ;Deskr. Block Adresse abgelegt wird.
02DE'   2A 0051"                LD      HL,(DESADR)     ;Adresse ablegen
02E1'   DD 75 00                LD      (IX),L
02E4'   DD 74 01                LD      (IX+1),H

02E7'   DD 2A 0051"             LD      IX,(DESADR)             ;Task in den Zustand waiting versetzen.
02EB'   DD 36 02 04             LD      (IX+STATUS),WAITNG

02EF'   FD 7E 01                LD      A,(IY+PUTIDX)   ;Index neu ermitteln
02F2'   3C                      INC     A               ;Index =(Index+1) modulo (NTASKS-1)
02F3'   06 04                   LD      B,NTASKS-1
02F5'   CD 024B'                CALL    MOD
02F8'   FD 77 01                LD      (IY+PUTIDX),A   ;nach Index zurueckspeichern
02FB'   3E 00                   LD      A,NOERR         ;Rueckmeldung kein Fehler
02FD'   CD 0173'                CALL    SETA
0300'   C3 0099'                JP      SELTSK          ;Naechste anstehende Task auswaehlen und
                                                        ;starten.
```

```
Systemkern BS0                    (180685)        MACRO-80 3.43   27-Jul-81       PAGE    I-18
Semaphoren Verwaltung


  0303'   FD 35 00                FREI:   DEC     (IY+COUNT)      ;Counter dekrementieren
  0306'   3E 00                           LD      A,NOERR         ;Return Status OK setzen
  0308'   CD 0173'                        CALL    SETA

  030B'   C3 0099'                        JP      SELTSK          ;Weiter mit naechster Task


  030E'                   +       WERRET: SPRELO
                          +               RRELOD
  0326'   3E 20                           LD      A,RSNOSE
  0328'   FB                              EI
  0329'   C9                              RET
                                          PAGE



                                  ;
                                  ;
                                  ;
                                  ;*****************************************************************
                                  ;
                                  ; SIGNAL  Signal senden
                                  ;
                                  ;       Akku muss Sema# enthalten
                                  ;
                                  ;       Eingang aus Interruptebene bei SIGFI.
                                  ;       Beachte: Register werden ueber SIGFI nicht gesichert!
                                  ;
                                  ;
                                  ;
                                  ;*****************************************************************
                                  ;
032A'   F3                        SIGNAL: DI              ;Ueblicher Vorspann
                          +               RSAVE           ;Eingang von einer Task
                          +               SPSAVE
0347'   ED 7B 0045*                       LD      SP,(SYSSTK)
034B'   CD 0375'                          CALL    SIGN1   ;Fuehre Signal Funktion aus

034E'   DA 0359'                          JP      C,SIGER ;Fehlermeldung
0351'   3E 00                             LD      A,NOERR
0353'   CD 0173'                          CALL    SETA    ;sonst OK
0356'   C3 0099'                          JP      SELTSK          ;naechste ready Task starten


0359'                     +       SIGER:  SPRELO
                          +               RRELOD
0371'   3E 20                             LD      A,RSNOSE        ;Fehlermeldung erzeugen
0373'   FB                                EI
0374'   C9                                RET
```

```
0375'                   SIGFI:                                  ;Eingang von der Interruptebene/Treiber
0375'  CD 0244'         SIGN1:  CALL    VALID
0378'  DA 03BB'                 JP      C,SERRET                ;Fehlerreturn

037B'  CD 022C'                 CALL    IYPOS                   ;IY auf Sema Block positionieren.

037E'  FD 7E 01                 LD      A,(IY+PUTIDX)           ;If Warteschlange leer then LEER
0381'  FD BE 02                 CP      (IY+GETIDX)
0384'  CA 03B6'                 JP      Z,LEER

                        ; Warteschlange ist nicht leer --> naechste Task aus Schlange holen und
                        ; ready setzen.

0387'  26 00                    LD      H,0H            ;IX wie bei WAIT auf die Adresse positionieren
0389'  FD 6E 02                 LD      L,(IY+GETIDX)   ;welche die Desk. Block Adresse enthaelt.
038C'  29                       ADD     HL,HL           ;Index=Index*2
038D'  FD E5                    PUSH    IY
038F'  DD E1                    POP     IX
0391'  E5                       PUSH    HL
0392'  D1                       POP     DE              ;DE=HL
0393'  DD 19                    ADD     IX,DE
0395'  11 0003                  LD      DE,SBAS         ;Offset Addieren
0398'  DD 19                    ADD     IX,DE
                                                        ;IX zeigt jetzt auf Desk. Adresse
039A'  DD 6E 00                 LD      L,(IX)          ;Desk. Adresse nach HL bringen
039D'  DD 66 01                 LD      H,(IX+1)
03A0'  E5                       PUSH    HL
03A1'  DD E1                    POP     IX              ;IX zeigt jetzt auf Desk. BLock

03A3'  DD 36 02 02              LD      (IX+STATUS),READY       ;Task wird ready

03A7'  FD 7E 02                 LD      A,(IY+GETIDX)           ;Index verwalten (wie bei WAIT)
03AA'  3C                       INC     A                       ;Index=(Index+1) modulo (NTASKS-1)
03AB'  06 04                    LD      B,NTASKS-1
03AD'  CD 024B'                 CALL    MOD
03B0'  FD 77 02                 LD      (IY+GETIDX),A           ;Im Sema Block sichern
03B3'  C3 03B9'                 JP      SIGRET

03B6'  FD 34 00         LEER:   INC     (IY+COUNT)              ;Nur Counter erhoehen

03B9'  A7               SIGRET: AND     A                       ;CARRY=0
03BA'  C9                       RET

03BB'  37               SERRET: SCF
03BC'  3E 20                    LD      A,RSNOSE        ;Error Kode No Sema
03BE'  C9                       RET

                        ;
                        ;
                                PAGE
```

```
                        SUBTTL Nachrichtenpufferverwaltung
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; Ringpuffer (Daten) Verwaltung f. BS0
                        ;
                        ; Dient zur Intertaskkommunikation
                        ;
                        ;*****************************************************************
                        ;
                        ;
                        ;
0000                    PIDXP   EQU     0       ;Offset im Pufferblock fuer den Index (put)
0001                    GIDXP   EQU     1       ; "     "       "       "  "    "   (get)
0002                    PBAS    EQU     2       ; "     "       "       "  "  Beginn der Daten
000E                    PUFOFF  EQU     (MAXENT*PUFLNG)+2       ;Zwischenraum zwischewn den einzelnen Bloecken


03BF'                           DSEG
00B6*                   PUFERS: DS      NPUF*PUFOFF             ;Raum f. Speicherung der Daten
00E0*                           CSEG
                        ;
                        ;
                        ;
                        ; Interne Prozeduren der Ringpuferverwaltung
                        ;
                        ;-----------------------------------------------------------------
                        ; VALIDP
                        ;-----------------------------------------------------------------
                        ;
                        ; prueft ab ob die Pufer# gueltig (im Bereich 0..npuf-1) ist
                        ;
                        ; <A>=Puffer#     --> Carry=1 dann Falsch
                        ;
03BF'   FE 03           VALIDP: CP      NPUF    ;if gueltig then reset Carry
03C1'   3F                      CCF
03C2'   D0                      RET     NC      ;Return falls alles klar!
03C3'   3E 30                   LD      A,RSNOBU        ;Fehlerkode No Buffer
03C5'   C9                      RET
                        ;
                        ;
                        ;
                        ;-----------------------------------------------------------------
                        ; IYPOSP
                        ;-----------------------------------------------------------------
                        ;
                        ; IY auf Eintrag positionieren
                        ;
                        ; <A>=Puffer#
                        ;
03C6'   C5              IYPOSP: PUSH    BC
03C7'   D5                      PUSH    DE
03C8'   47                      LD      B,A             ;Akku enthaelt Nummer
```

```
Systemkern BS0                  (180685)        MACRO-80 3.43   27-Jul-81       PAGE    1-22
Nachrichtenpufferverwaltung

  03C9'   04                            INC     B
  03CA'   FD 21 00B6'                   LD      IY,PUFERS       ;In IY die Adresse des Blocks berechnen
  03CE'   11 000E                       LD      DE,PUFOFF
  03D1'   10 03                 MLP:    DJNZ    MLADDP          ;A mal wird der Offset (PUFOFF) zu PUFERS
  03D3'   C3 03D8'                      JP      MLENDP          ;addiert
  03D6'   FD 19                 MLADDP: ADD     IY,DE
  03D8'   C3 03D1'                      JP      MLP
  03DB'   D1                    MLENDP: POP     DE              ;Ende Return
  03DC'   C1                            POP     BC
  03DD'   C9                            RET
                                ;
                                ;
                                        PAGE



                                ;
                                ;
                                ;*****************************************************************
                                ;
                                ; INIPUF     Initialisieren der zugehoerigen Semaphorvariablen
                                ;
                                ;*****************************************************************
                                ;
                                ; Aufruf bei Power Up
                                ;
03DE'   F3                      INIPUF: DI
                        +               RSAVE           ;Reg Sichern

03EF'   06 03                           LD      B,NPUF                  ;Anzahl der Puffer in B zaehlen
03F1'   3E 02                           LD      A,NSEMAS-1              ;bei Nsemas gehts los

03F3'   0E 03                   CONTIN: LD      C,MAXENT                ;Sema Leer = Maxent
03F5'   3C                              INC     A
03F6'   F5                              PUSH    AF                      ;Akku sichern
03F7'   CD 025E'                        CALL    ININCH                  ;Sema Leer initialisieren
03FA'   F3                              DI
03FB'   F1                              POP     AF

03FC'   3C                              INC     A                       ;auf naechste Sema gehen
03FD'   0E 00                           LD      C,0                     ;und mit Null init.
03FF'   F5                              PUSH    AF
0400'   CD 025E'                        CALL    ININCH
0403'   F3                              DI
0404'   F1                              POP     AF


0405'   10 EC                           DJNZ    CONTIN                  ;naechste Puffersemas verarbeiten

                        +               RRELOD
0417'   C9                              RET
```

```
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; SEND
                        ;
                        ;*****************************************************************
                        ;
                        ;
                        ; Eintrag eines Datenelementes im angegebenen Ringpuffer
                        ;
                        ; <A>=Puffer#      <BC>=Pointer auf die Information
                        ;
0418'                 +  SEND:   RSAVE
0428'   CD 03BF'                 CALL    VALIDP          ;Nummer ok?
042B'   DA 0492'                 JP      C,SENER         ;Jump if error

042E'   F5                       PUSH    AF              ;Puffernummer auf dem Stack sichern
                        ;
                        ; wait auf Sema ausfuehren; Prozess wuerde sich hier verzoegern, falls
                        ; der Puffer momentan voll waere.
                        ;
                        ; Sema# berechnen
                        ; SEMA# = NSEMAS+(<A>*2)
042F'   CB 27                    SLA     A               ;Akku=Akku*2
0431'   C6 03                    ADD     A,NSEMAS                ;A enthaelt jetzt Sema #
0433'   CD 0299'                 CALL    WAIT

0436'   FE 00                    CP      NOERR           ;If RS=ok then weiter
0438'   C2 0000*                 JP      NZ,CRASH        ;else Crashkode ausfuehren

043B'   F1                       POP     AF              ;Puffer# wieder restaurieren
043C'   CD 03C6'                 CALL    IYPOSP          ;IY in Position auf den Pufferblock bringen
043F'   F5                       PUSH    AF              ;und Nummer wiederv sichern

0440'   F3                       DI                      ;enter critical region

0441'   21 0000                  LD      HL,0            ;Ziel Adresse des Transfers berechnen
0444'   11 0004                  LD      DE,PUFLNG       ; DE=IY+PBAS+(PIDXP*PUFLNG)
                                                         ;zuerst PIDXP*PUFLNG

0447'   FD 7E 00                 LD      A,(IY+PIDXP)
044A'   FE 00                INDXP: CP   0               ;Multipl. durch Addition
044C'   CA 0454'                 JP      Z,BASIYS
044F'   19                       ADD     HL,DE
0450'   3D                       DEC     A               ;Akku runterzaehlen
0451'   C3 044A'                 JP      INDXP

0454'   11 0002              BASIYS: LD  DE,PBAS         ;PBAS addieren
0457'   19                       ADD     HL,DE
```

```
Systemkern BS0                (180685)                                    PAGE    1-24
Nachrichtenpufferverwaltung

 0458'  FD E5                        PUSH   IY
 045A'  D1                           POP    DE              ;Blockbasis IY addieren
 045B'  19                           ADD    HL,DE

 045C'  E5                           PUSH   HL
 045D'  D1                           POP    DE              ;DE zeigt jetzt auf die Zieladresse im Pufferblock

 045E'  69                           LD     L,C             ;Quelladresse aus dem BC Reg. holen
 045F'  60                           LD     H,B             ;und ins HL schaffen

 0460'  01 0004                      LD     BC,PUFLNG       ;Anzahl der zu transferierenden Bytes ins BC

 0463'  ED B0                        LDIR                   ;Transport
                             ;
                             ;
                             ;
                             ; Index=Index+1  modulo npuf-1
                             ;
 0465'  FD 7E 00                     LD     A,(IY+PIDXP)
 0468'  3C                           INC    A                       ;Index erhoehen
 0469'  06 03                        LD     B,MAXENT
 046B'  CD 024B'                     CALL   MOD
 046E'  FD 77 00                     LD     (IY+PIDXP),A            ;Ergebnis zuruckspeichern
                             ;
 0471'  FB                           EI                             ;leave critical region

                             ;
                             ; signal auf Sema senden um evtl wartende Prozesse weiterlaufen zulassen
                             ;
 0472'  F1                           POP    AF                      ;PUFFER# holen
 0473'  CB 27                        SLA    A                       ;und aus ihr die Sema # berechnen
 0475'  C6 04                        ADD    A,NSEMAS+1
 0477'  CD 032A'                     CALL   SIGNAL

 047A'  FE 00                        CP     NOERR                   ;falls Return mit Fehler dann Crash
 047C'  C2 0000*                     JP     NZ,CRASH

                        +            RRELOD                         ;Register rueckladen
 048F'  3E 00                        LD     A,NOERR         ;Ret. Status=ok
 0491'  C9                           RET

                             ;
                             ; Error Retyurn
 0492'                  +    SENER:  RRELOD
 04A2'  3E 30                        LD     A,RSNOBU                ;Puffer# falsch
 04A4'  C9                           RET
                             ;
                             ;
                             ;
                             ;
                                     PAGE
```

```
                        ;
                        ;
                        ;
                        ;
                        ;*****************************************************************
                        ;
                        ; RECEIV
                        ;
                        ;*****************************************************************
                        ;
                        ; Info. Element aus dem Puffer entnehmen
                        ;
                        ; <A>=Puffer#      <BC>=Poineter auf den zur Verfuegung gestellten Puffer
                        ;
                        ;
04A5'                +  RECEIV: RSAVE

04B5'  CD 03BF'                 CALL    VALIDP          ;# Ok ?
04B8'  DA 051D'                 JP      C,RECER         ;If Not OK then Error

04BB'  F5                       PUSH    AF              ;Puffer# sichern

                        ;
                        ; WAIT auf Sema ausfuehren, falls Prozess aus einem leeren Puffer lesen
                        ; will wird er hier verzoegert
                        ;
                        ;
                        ; Sema# berechnen
                        ; Sema#=NSEMAS+(<A>*2)+1
                        ;
04BC'  CB 27                    SLA     A                       ;Akku mal 2
04BE'  C6 04                    ADD     A,NSEMAS+1
04C0'  CD 0299'                 CALL    WAIT

04C3'  FE 00                    CP      NOERR                   ;Error bei Sema?
04C5'  C2 0000*                 JP      NZ,CRASH                ;falls ja dann Crashkode ausf.

04C8'  F1                       POP     AF                      ;Nummer wieder holen
04C9'  CD 03C6'                 CALL    IYPOSP                  ;und IY positionieren
04CC'  F5                       PUSH    AF                      ;Nummer sichern

04CD'  F3                       DI                              ;enter critica;l region

                        ;
                        ;
                        ; HL=IY+PBAS+(INDEX(get)*PUFLNG)
04CE'  21 0000                  LD      HL,0                    ;zuerst GIDXP*PUFLNG
04D1'  11 0004                  LD      DE,PUFLNG
04D4'  FD 7E 01                 LD      A,(IY+GIDXP)            ;Index holen
04D7'  FE 00            INDXG:  CP      0                       ;Puflng wird A mal addiert
04D9'  CA 04E1'                 JP      Z,BASIYR
04DC'  19                       ADD     HL,DE                   ;Addieren
04DD'  3D                       DEC     A
04DE'  C3 04D7'                 JP      INDXG
```

```
Systemkern BS0                  (180685)        MACRO-80 3.43   27-Jul-81       PAGE    1-27
Nachrichtenpufferverwaltung

                                ;
 04E1'   11 0002                BASIYR: LD      DE,PBAS                 ;PBAS dazu
 04E4'   19                             ADD     HL,DE

 04E5'   FD E5                          PUSH    IY
 04E7'   D1                             POP     DE                      ;+Block Basis Adresse

 04E8'   19                             ADD     HL,DE

 04E9'   59                             LD      E,C
 04EA'   50                             LD      D,B                     ;DE = Zieladresse

 04EB'   01 0004                        LD      BC,PUFLNG               ;Laenge ins BC

 04EE'   ED B0                          LDIR                    ;Transfer


 04F0'   FD 7E 01                       LD      A,(IY+GIDXP)            ;Index=Index+1 Mod npuf-1
 04F3'   3C                             INC     A
 04F4'   06 03                          LD      B,MAXENT
 04F6'   CD 024B'                       CALL    MOD
 04F9'   FD 77 01                       LD      (IY+GIDXP),A            ;zuruck speichern

 04FC'   FB                             EI                              ;leave critical region

                                ;
                                ;
                                ;
                                ; signal auf sema senden
                                ; evtl. wartende sender werden angestossen
                                ;
 04FD'   F1                             POP     AF              ;Pufer# holen
 04FE'   CB 27                          SLA     A               ;Sema# berechnen

 0500'   C6 03                          ADD     A,NSEMAS
 0502'   CD 032A'                       CALL    SIGNAL

 0505'   FE 00                          CP      NOERR           ;wenn <>0 --> Sytemfehler
 0507'   C2 0000*                       JP      NZ,CRASH

                        +               RRELOD
 051A'   3E 00                          LD      A,NOERR         ;RS=ok
 051C'   C9                             RET

 051D'                  +       RECER:  RRELOD
 052D'   3E 30                          LD      A,RSNOBU        ;Puffer# falsch
 052F'   C9                             RET

                                        END
```

```
                                .Z80
                                PUBLIC BASIS,TREIB2,CRASH
                                EXT     ENDE,ACTIVE,DELAY,WAIT,SIGNAL,INISEM,SIGFI
                        ;
                        ; Demonstrationstasks fuer BS0 Kern
                        ;
                        ; Hier soll nur ein Beispiel gegeben werden, wie
                        ; die TCBs sowie die Grundtabelle BASIS aufgebaut werden kann.
                        ; Die angeschlossenen Tasks fuehren nichts Vernuenftiges aus, sondern
                        ; zeigen lediglich einige BS0 Aufrufe.
                        ;
                        ; TASK1: Diese Task lauft zyklisch  alle zwei Sekunden.
                        ;        Sie schaltet einen Ausgang ein, hier eine Kontrollampe und
                        ;        stoesst die Task2 an. Nach dem Aktivieren der Task2 wird Task1
                        ;        um eine Sekunde verzoegert und danach die Lampe wieder ausgeschaltet.
                        ;
                        ;
                        ; TASK2: Nach ihrer Aktivierung durch T1 schaltet Task2 den Ausgang 5 ein,
                        ;        wartet eine halbe Sekunde und schaltet Ausgang 5 wieder aus.
                        ;        Die Task wird nur durch T1 aktiv!
                        ;
                        ;
                        ;
0000'                           DSEG

0000*   02              BASIS:  DB      2       ;ANZAHL DER TASKS
0001*   0033*                   DW      TSKD1   ;POINTER AUF TASKDESKRIPTOREN
0003*   006B*                   DW      TSKD2

                                ;
0005*                           DS      20H     ;PRIVATER STACK DER TASK 1
0025*   0000            STP:    DW      0       ;REGISTER STATI BEIM BEGINN
0027*   0000                    DW      0
0029*   0000                    DW      0
002B*   0000                    DW      0
002D*   0000                    DW      0
002F*   0000                    DW      0
0031*   0000'                   DW      TASK1   ;HIER WIRD SPAETER DER PC GEHOLT
0033*   0025*           TSKD1:  DW      STP     ;STACK POINTER GESICHERT
0035*   08                      DB      00001000B       ;STATUS HIER TERMINATED
0036*   0000'                   DW      TASK1           ;ANFANGS ADRESSE VON TASK1
0038*   C8                      DB      200D            ;TIMER
0039*   C8                      DB      200D            ;RELOAD TIMER FROM HERE
003A*   00                      DB      0               ;TIMER WIRD VON DELAY GEBRAUCHT
003B*   5431                    DW      'T1'            ;TASK NAME
                        ;
                        ;
003D*                           DS      20H     ;PRIVATER STACKBEREICH
005D*   0000            STP2:   DW      0
005F*   0000                    DW      0
0061*   0000                    DW      0
0063*   0000                    DW      0
0065*   0000                    DW      0
0067*   0000                    DW      0
0069*   0016                    DW      TASK2
006B*   005D*           TSKD2:  DW      STP2            ;STACK POINTER
```

```
006D*   08                      DB      00001000B   ;STATUS HIER TERMINATED
006E*   0016'                   DW      TASK2       ;RESTART ADRESSE FUER TASK
0070*   01                      DB      001D        ;TIMER
0071*   00                      DB      000D        ;ZYKLUSZEIT
0072*   00                      DB      0           ;RESERVIERT FUER DELAY
0073*   5432                    DW      'T2'        ;TASKSNAME
                        ;
                        ;

0075*                           CSEG
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
0000'   3E 01           TASK1:  LD      A,01        ;EINSCHALTEN EINER KONTROLLAMPE
0002'   D3 01                   OUT     (01),A      ;DURCH AUSGABE AUF EINEN PORT

0004'   01 5432                 LD      BC,'T2'     ;TASK2 AKTIVIEREN
0007'   CD 0000*                CALL    ACTIVE

000A'   3E 64                   LD      A,100D      ;EINE SEKUNDE VERZOEGERN
000C'   CD 0000*                CALL    DELAY

000F'   3E 00                   LD      A,00        ;AUSSCHALTEN DER KONTROLLAMPE
0011'   D3 01                   OUT     (01),A      ;DURCH AUSGABE EINER NULL

0013'   C3 0000*                JP      ENDE        ;TASK ENDE
                        ;
                        ;
0016'   3E 32           TASK2:  LD      A,50D       ;EINE HALBE SEKUNDE WARTEN
0018'   CD 0000*                CALL    DELAY

001B'   3E 01                   LD      A,01        ;KONTROLLAMPE 5 EINSCHALTEN
001D'   D3 05                   OUT     (05),A

001F'   3E 32                   LD      A,50D       ;EINE HALBE SEKUNDE BRENNEN LASSEN
0021'   CD 0000*                CALL    DELAY

0024'   3E 00                   LD      A,00        ;UND WIEDER AUSSCHALTEN
0026'   D3 05                   OUT     (05),A

0028'   C3 0000*                JP      ENDE        ;TASK ENDE
                        ;
                        ;
                        ;
002B'   76              CRASH:  HALT                ;CRASH WIRD NUR IN FATALEN FEHLERFAELLEN
                                                    ;ANGESPRUNGEN, HIER MUESSTE EINE ENTSPRECHENDES
                                                    ;FEHLERHANDLING ERFOLGEN
                        ;
                        ;
002C'   C9              TREIB2: RET                 ;DUMMY FUER INTERRUPT VECTOR 2 HIER MUESSTE
                                                    ;EIN INTERRUPT TREIBER STEHEN.
                        END
```

Sachverzeichnis

A
ACTIVE 22, 26
allocate 76ff.
Ankeradresse 24, 39
Anwenderprogramme 12
Assembler 21

B
BASIS 24
Befehlszähler 13, 21
Betriebs|mittel 28, 33
–systemkern 11ff., 23, 37
Bibliotheken 83
Block 76ff.
BOX 57ff.
BSO 23ff., 42f., 59, 61, 68

C
Compiler 21
CONSOLE 46, 48, 56
CREATE 23

D
Dateiverwaltung 10
Daten
–pufferung 45
–transport 45ff.
DELAY 22, 26
delayed 22f.
Deskriptorblock 16ff., 23, 39, 41
Disable Interrupt 30
Dispatcher 13ff., 17, 19ff.
Durchlaufzeit 20

E
empty 49
Enable Interrupt 30

F
Fehler
–behandlung 71ff.
–erkennung 72
–kategorie 74, 75
–korrektur 75
–toleranz 71
FIFO 34f., 55f.
Flag 29ff., 35, 43
full 49

G
Gerätetreiber 66
GETIDX 42
Grund|initialisierung 38
–takt 18, 27

H
HALT 73
Hintergrundmonitor 84
Hochsprache 21

I
Idle 16
Implementierungssprache 21
INISEM 42
Interrupt 13, 62ff., 66
–handler 64f., 66
–vektorisierter 63, 66
–verwaltung 62ff.
–sperrung 23
Intervall 14

K
Kernprozedur 14, 21ff.
KOMMAND 82f.
Kritische Bereiche 31f.

L
Lagerverwaltung 11ff.
LIFO 55

M
Mailbox 35ff., 46, 50, 56, 57, 67, 73, 79, 81f.
Map 76ff.
Maschinencode 11
Mehrbenutzersystem 20

Mehrfachzugriff 28
Mehrprogrammbetriebssystem 10, 16
MOD 43

N
Nachricht 46
Nachrichten|schlange 54
–Queue 53
NIL 54
NMI 64
no wait 48f.
NSEMAS 42

P
Pool 53f., 76ff.
POWUP 25
Prioritäten 16f., 55f.
Produzent 45
Prozeß 11ff., 34
–deskriptor 16ff., 39, 41, 53
–identifikation 18
–kommunikation 57ff.
–kooperierende 28
–suspendierung 19, 23
–verwaltung 16
–zustand 17
Prozessorzeit 32
Puffer 45ff., 55f., 68
–verwaltung 46, 48, 55
PUTIDX 42

Q
Queue 53, 55

R
ready 14ff., 64
Reaktionszeit 62f.
Realzeit 63
receive 46f., 50, 54
reentrante Prozeduren 81, 83
Reihumverfahren 14
release 78ff.
Rendezvous 59
Ring 50, 52
Ringpuffer 50ff.
Round-Robin 14, 16, 19f.
Rückkehradresse 62
Rücksprungadresse 25
running 14ff., 64

S
Scheduler 15, 17, 19ff.
Schwarzes Brett 35
SELTSK 26
Semaphoren 28, 33ff., 39ff., 46
send 46f., 50, 54
signal 33ff., 37, 68
Speicherverwaltung 53, 76ff.
Stack 21
–pointer 16ff.
Synchronisation 30, 33, 37, 47, 68
System
–kern 46
–prozeduren 21ff., 31
–prozeß 66

T
Task 11, 22f., 34
–neustart 24
Telefon 35
terminated 15f.
Textverarbeitung 11ff.
TICK 26, 68
Time-Sharing 14
Trap 64
Treiber 66

U
UART 66f.
Uhr
–takt 13f.
–zeit 20
Unterbrechende Steuerung 19f.

V
Verklemmung 80

W
wait 33ff., 37
Warteschlange 34f., 37f., 53f.

Z
Zähler 37
Zeit
–interrupt 13
–intervall 11
Zeitraster 13f.
Zugriffsrechte 19
Zustandsgraph 14f.
Zykluszeit 14

Zilker
Praxis des Multitasking

Dem Programmierer wird hier der richtige Weg durch die vielfältigen Möglichkeiten gezeigt, die Multitasking bietet: Nämlich die quasi gleichzeitige Abarbeitung mehrerer Programme. Alle wesentlichen Praxispunkte sind dabei berücksichtigt; die beschriebenen Methoden lassen sich gezielt verwirklichen.

Angesprochen werden unter anderen: Spezifische Probleme, wie das Verwalten der Peripherie, eine exakte zeitliche Koordinierung per Interrupt oder Polling. Diese Techniken werden praxisgerecht und in die Tiefe gehend abgehandelt. Dazu kommen Kommunikationsmöglichkeiten unter den einzelnen Tasks, wie Semophoren und Wortschlangen.

Der Leser ist nach der Lektüre dieses Bandes in der Lage, die Grenzen des Singletasking zu überschreiten und selbständig einen Multitasking-Kern zu implimentieren.

Der Autor ist Dipl.-Informatiker und beschäftigt sich schwerpunktmäßig mit Microprozessor-Steuerungen, Hochsprachen und Systemdesign.

03800
9 783772 385612
ISB N 3-7723-8561-3 DM +038.00

Franzis'